Proposed Architecture for Call Completion Supplementary Services in Asterisk

Version 1.0

Table of Contents

1. CCSS Introduction

CCSS stands for the Call Completion Supplementary Services. In Asterisk, CCSS support will include both CCBS and CCNR, which represent CCSS for when the callee is busy (Busy Subscriber) or not available (No Response). For CCSS to be successful, it is important that the interface be granular enough to support multiple different protocols without having to take special precautions to support each one. It is important that each component has well-defined roles so that when a new component is created, it is known exactly what it should do.

Call completion support in Asterisk is composed of three components. First, there is the call completion agent. The call completion agent acts on behalf of the calling party. The agent subscribes to CCBS/CCNR and is notified by the call completion core when it is possible for CCBS/CCNR to be completed. Second, there is the call completion monitor. The call completion monitor's job is to watch the state of the called party. It is the job of the monitor to notify the call completion core when the called party is ready to receive a CCBS/CCNR call. The final component of the call completion system is the call completion core. The core's job is to maintain the state of call completion tasks in the system and act as a liaison between the agent and monitor.

2. The CCSS Core

2.1 Overview

The core can be seen as the “controller” for all call completion services in Asterisk. Its job is to maintain state of all call completion transactions, answer user queries regarding general call completion statistics, and notify agents and monitors of state changes. The core is also responsible for maintaining a queue of active CCBS/CCNR requests. The implementation details of this queue are discussed further in the section 4.2.

The core obtains its configuration from the file `callcompletion.conf`. Currently, the only global CCSS option is for allowing the administrator to place a global limit on the number of outstanding CC requests that are active. Effectively, this option controls the global number of actively running monitor state machines. These state machines may be generic monitor instances, or technology specific monitors.

•maxccssrequests = <integer>

◦This option controls how many active CCSS requests may be pending on this system. By default, there is no limit put in place.

2.2 The CCSS state machine

The core implements a state machine to facilitate CCBS/CCNR. Because we wish for the core to interact with both the generic and protocol-specific versions of the agent and monitor, the state machine for the core is kept simple. While the state machine is maintained in the core, most state changes are requested by the agent and monitor components. The core actually implements the state change and then notifies the agent and monitor components of the change. Below are descriptions of each of the CCBS/CCNR states and the core's role during each. To get a clearer picture of the operation of the state machine, a state change diagram is included in Figure 1. Note that the diagram does not cover all reasons that may result in `CC_FAILED`. For instance, it does not take into account low level problems such as running out of memory. Also note that CCNR is not mentioned in the diagram. This is for the sake of brevity; the diagram applies just as much for CCNR as it does for CCBS.

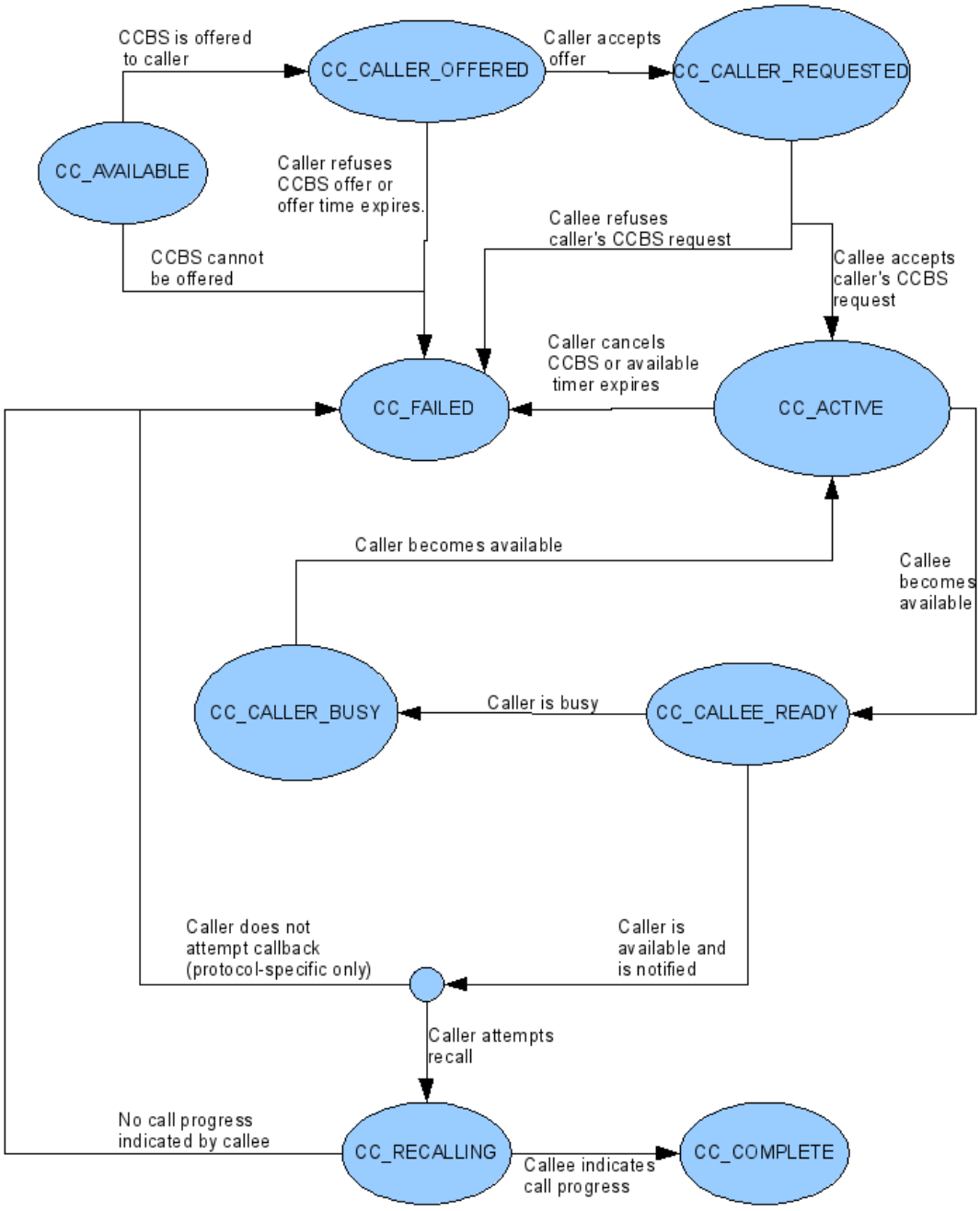

CCBS is offered to caller
CC_CALLER_OFFERED
Caller accepts offer
CC_CALLER_REQUESTED
CC_AVAILABLE
Caller refuses CCBS offer or offer time expires.
CCBS cannot be offered
Callee refuses caller's CCBS request
Callee accepts caller's CCBS request
Caller cancels CCBS or available timer expires
CC_FAILED
CC_ACTIVE
Caller becomes available
Callee becomes available
CC_CALLER_BUSY
Caller is busy
CC_CALLEE_READY
Caller does not attempt callback (protocol-specific only)
Caller is available and is notified
Caller attempts recall
No call progress indicated by callee
CC_RECALLING
Callee indicates call progress
CC_COMPLETE

Figure 1

The core is responsible for starting the CCSS state machine after a call in which it has been determined that CCBS/CCNR is possible on at least one outbound call leg. During an outbound call, it is the job of the outbound Asterisk channel driver to decide whether CCSS will be possible. In addition to checking configuration options specific to the device being dialed, the channel driver also will consult with the CCSS core to ensure that a new CCSS transaction will not exceed the configured `maxccssrequests` and that there is not already an outstanding request between the two parties. Once the channel driver has determined that outbound CCBS or CCNR is possible, the channel driver will signal the information to the controlling application using an `AST_CONTROL_CC` frame. This information will then be made available to the administrator via the `CC_AVAIL` channel variable that may be inspected in the dialplan. Using this variable, the administrator then can know that CCSS is possible for the call and can determine if CCBS/CCNR should be offered. For an example which uses this variable, please see section 5.3.

In addition, the core keeps a record of the failed call, specifically remembering the calling device, and all called extensions/devices. This is done so that when the calling party attempts to request CCBS/CCNR, the call completion agent may query the core for information necessary to complete the request. From this information, the agent can determine if the request for CCBS/CCNR is legitimate.

Below are descriptions of each of the states in the state machine, as well as the core's role during each.

2.2.1 CC_AVAILABLE

The core starts the CCSS state machine in this state and saves information for this call, such as who the calling and called parties are and which of CCBS or CCNR was offered by each called party. The core also will create the appropriate CCSS agent during this state. The purpose of this state is to wait for the newly created agent to alert the core that it has offered the caller CCBS/CCNR.

Note that while the determination of the possibility of utilizing CCSS may be made during the call, this state will not be entered until the outbound Dial application has completed.

2.2.2 CC_CALLER_OFFERED

This state is reached when the agent alerts the core that a caller has been offered CCBS/CCNR. The purpose of this state is to wait for the agent to alert the core that the caller has accepted the CCBS/CCNR offer by requesting the appropriate service.

2.2.3 CC_CALLER_REQUESTED

This state is reached when the agent alerts the core that a caller has accepted a CCBS/CCNR offer. When the core changes to this state, it should create the necessary structures for the monitor side of the CCSS transaction. The purpose of this state is to wait for the monitor to alert the core that it has received an acknowledgment of its outbound CCNR/CCBS request.

2.2.4 CC_ACTIVE

This state is reached when the monitor alerts the core that it has received a positive acknowledgment for the outbound CCNR/CCBS request it had made earlier. When the core is alerted that this state has been reached, it should increment a counter whose upper bound is the `maxccssrequests`. The purpose of this state is to wait for the monitor to alert the core that the called party has become available.

2.2.5 CC_CALLEE_READY

This state is reached when the monitor alerts the core that its callee is now available to receive a call from the CCBS/CCNR queue. If there are multiple parties waiting on the same person, then it is the core's duty at this point to choose which of the waiting parties will be notified that the person has become available. During this state, the core awaits a response from the agent that either the caller is busy, or that the caller is not busy and is attempting its recall.

2.2.6 CC_CALLER_BUSY

This state is reached if the agent alerts the core that the caller which had requested CCBS/CCNR is currently unavailable and unable to complete his previously submitted request. During this state, the core waits for the agent to tell it that the caller has become available again.

2.2.7 CC_RECALLING

This state is reached when the agent alerts the core that the caller is attempting a CCBS/CCNR recall. The purpose of this state is to wait for a call progress message to be received on the outbound leg of the recall. The change request will most likely be made from the channel driver receiving the progress response.

2.2.8 CC_COMPLETE

This state is reached when the core is alerted that call progress has been made on the recall. The core should alert the monitor and agent of this change and free any of its resources for this call, including its saved information regarding the initial failed call. The core also must decrement its counter of total CCSS transactions.

2.2.9 CC_FAILED

This state is reached if at any point during the call completion process, the monitor or agent reports that CCBS/CCNR may not be completed. Note that any stimulus from an agent or monitor to enter this state may contain an accompanying reason for debugging and statistical purposes. All other memory allocated, including the saved call information which was recorded when the original call failed, will be freed. The core also must decrement its counter of total CCSS transactions.

3. The CCSS Agent

3.1 Overview

The CCSS agent is responsible for performing actions on behalf of the party requesting CCBS/CCNR as well as sending and receiving signaling to and from the party requesting CCBS/CCNR. There are two classifications of CCSS agents, generic and protocol-specific. A protocol-specific CCSS agent is used for callers whose channel protocol natively supports CCBS/CCNR. The generic CCSS agent is used when a caller's channel protocol does not natively support CCBS/CCNR.

The CCSS agent is responsible for one timer, which is the offer timer. Once CCBS/CCNR has been offered to the caller, it is the responsibility of the agent to cancel the offer

if a request from the caller is not received after some configurable period of time. The length of the offer timer should be configurable per device. It should also be recommended to administrators that the offer timer be set to a smaller value for phones directly connected to the server as opposed to trunking interfaces.

3.1.2 Configuration options

All CCSS agents have configuration options associated with their operation. These options may be configured on any channel driver configuration, either in the general section or per-device.

*`cc_agent_policy` – Defines how and if Asterisk should attempt to create an agent for a given channel. There are three possible options:

- `never` – Never offer call completion services on this channel.
- `native` – Offer CC services using native messaging as defined by the channel's protocol. This option may be used if you know that the device connecting to the Asterisk server supports the signaling involved in call completion transactions.
- `generic` – Use a generic CCSS agent for the call. This option should only be used for phone configurations (as opposed to configuration for devices which are actually trunks). For more information on generic CCSS agents, see section 3.3.

*`cc_offer_timer` – The number of seconds we should wait after offering CC services until we remove the record of the call from the system.

*`cc_max_agents` – The maximum number of CCSS agents which may be allocated for this channel. In other words, this number serves as the maximum number of CC requests this channel is allowed to make.

3.2 The CCSS agent and the Core state machine

Below are general descriptions of CCSS agent behavior for each state of the core state machine.

3.2.1 CC_AVAILABLE

The CCSS agent is created during this state. While this state indicates that CCSS is

possible on the outbound leg, a CCSS agent does not have the authority to immediately offer CCBS or CCNR to the calling party. Even though CCSS may be available for the caller, it is the dialplan administrator who has the final say in whether CCSS is offered to the calling party. If the call ends with the administrator opting not to offer CCSS to the caller, then the agent should request for the core to change states to `CC_FAILED`. If the administrator decides to offer CCSS to the caller, then the agent should send whatever signaling is necessary to offer the service to the caller and then request that the core change states to `CC_CALLER_OFFERED`.

3.2.2 CC_CALLER_OFFERED

During this state, the agent's responsibility is to await a CCBS/CCNR request from the calling party. When this state is begun, the agent should start its `cc_offer_timer`. If the offer timer expires, the agent should request for the core to move to the `CC_FAILED` state.

If the calling party makes a CCBS/CCNR request while in this state, the agent should alert the core to change to the `CC_CALLER_REQUESTED` state.

3.2.3 CC_CALLER_REQUESTED

During this state, the agent's responsibility is to listen for any CCBS/CCNR cancellation attempts from the calling party. If the calling party should cancel, the agent should alert the core to change to the `CC_FAILED` state.

3.2.4 CC_ACTIVE

Protocol-specific agents will likely need to send an acceptance or acknowledgment response to a previously pending incoming CCBS/CCNR request. After that, the agent's duty is the same for this state as for `CC_CALLER_REQUESTED`.

3.2.5 CC_CALLEE_READY

During this state, the agent's first responsibility is to verify that the calling party is available. If the calling party is not available, then the agent should request that the core change to the `CC_CALLER_BUSY` state.

If the calling party is available, then the agent should alert the calling party that he may

attempt his CCBS/CCNR call. In some cases, this is done by having the agent actually place a call to the calling party. In others, this may be done using a protocol-specific message and waiting for the caller to place a call back. Once the caller has called the server back, the agent will initiate an outgoing call on behalf of the caller and request for the core to change state to `CC_RECALLING`.

The call that the agent makes on behalf of the caller requires some explanation. First, the agent will create a list of interfaces (obtained by querying the core) which should be dialed in order to ensure that the same devices which were dialed during the failed call are dialed again. These interfaces are placed in an ampersand-delimited list and set in the channel variable `CC_INTERFACES`. Then, the extension which was originally dialed by the caller is called again. It is recommended that the administrator places the contents of the `CC_INTERFACES` variable as the first parameter to `Dial` so that the outgoing calls can properly be recognized as being CC recalls. For an example of this, see section 5.3.

It may seem that the agent does quite a few operations during this state and that this state should be divided into smaller states. The problem is that there is no way to signal such state changes to downstream servers. Some implementations of CCSS agents may find it advantageous to define their own sub-states for `CC_CALLEE_READY` to denote progress in communicating with the calling party. Remember that such states should clearly be marked as implementation-specific and should not be communicated to the CCSS core. Also keep in mind that synchronization of such implementation-specific states between servers cannot be guaranteed the same way that synchronization of states in the core state machine can be.

3.2.6 CC_CALLER_BUSY

During this state, the agent's responsibility mirrors that of a device monitor (for more information on device monitors, see section 4.1.2). The agent needs to monitor the state of the caller. Once the caller becomes available, the agent should request that the core move to the `CC_ACTIVE` state.

3.2.7 CC_RECALLING

The agent has no specific responsibilities during this state.

3.2.8 CC_COMPLETE

The agent has no specific responsibilities during this state. Different implementations of agents may have cleanup operations or stats-keeping that need to be performed.

3.2.9 CC_FAILED

It may be necessary to notify the calling party that CCBS/CCNR is no longer available or has failed. In addition, the agent may need to perform some cleanup operations or stats-keeping.

3.3 The Generic CCSS agent

3.3.1 Overview

The generic CCSS agent is to be used when call completion supplementary services are desired, but the protocol used by the calling party does not support CCBS/CCNR. Because there is no protocol-specific method by which to offer CCBS/CCNR to the caller, the job falls instead to the Asterisk administrator to present the opportunity through a user interface, likely an IVR. The use of the CCSS agent is limited to phones connected directly to Asterisk, as the generic CCSS agent must be able to monitor the caller's availability, and doing so across multiple Asterisk servers is not supported.

There are two methods by which CCBS/CCNR may be requested when using the generic CCSS agent. With the first method, the caller is presented with the opportunity to request CCBS/CCNR while still on the phone after the initial call has failed. With the second method, the caller, after hanging up the initial failed call, dials a special extension, usually the appropriate vertical service code, to request CCBS/CCNR.

3.3.2 Applications and configuration options

3.3.2.1 Dialplan Applications

•`CallCompletionRequest()` – Allows for the calling party to request CCBS/CCNR. Because the agent and core's job is no different for CCBS and CCNR it is not necessary to offer separate applications for the two.

•`CallCompletionCancel()` – Allows for the calling party to cancel previously requested

CCBS/CCNR.

3.3.2.2 Dialplan Functions

*CALLCOMPLETION() - Allows for an administrator to read/write CCSS configuration parameters for a specific call. Valid parameters are any device-specific CCSS configuration options.

3.3.2.3 Manager Actions

•CallCompletionRequest – Request CCSS for the last call from a given device where CCNR or CCBS was offered.

•CallCompletionCancel – This action cancels an existing CCSS request.

3.3.2.4Manager Events

•CallCompletionRequested – This event is generated when a call completion request has been initiated. This event will include a request ID, so that later events can be associated with this one. This ID may also be used to cancel this CCSS request via a manager action.

•CallCompletionExpired – This event is generated when a call completion request has expired. It will include the request ID.

•CallCompletionAvailable – This event is generated when call completion services become available.

•CallCompletionCanceled – This event is generated when a CCSS request has been cancelled. The ID of the cancelled request is included.

3.3.2.5 Configuration Options

Since generic CCSS agents are just another type of CCSS agent, all configuration options from section 3.1.2 are applicable. In addition, one more configuration option is available which pertains strictly to generic CCSS agents.

•callback_macro – When the generic CCSS agent is notified by the core to call the calling party back, this macro will be run on the calling party's channel once he has answered the call. This will allow for an administrator to inject custom logic into the callback. For instance, a message notifying the caller that this is a CCBS/CCNR call could be played.

3.3.3 The Generic CCSS Agent and the Core state machine

Below are the specific duties of the generic CCSS agent for each state of the core state machine. Generic CCSS agents work a bit differently from protocol-specific agents in that they have multiple ways to offer the existence of CC services to the caller. The first way is one which aligns more with how protocol-specific CCSS agents behave. That is, a caller may notice that his call has failed, then hang up, and then pick up the phone and dial an extension so that CCBS/CCNR may be requested for his last dialed call. A second way that CC services may be offered to the caller when using a generic CCSS agent is to allow the caller to request the service while still on the phone from the initial failed call. Such a method would likely be accomplished by playing prompts to the user based on the `${DIALSTATUS}` of the failed call. The example in section 5.3 illustrates this method.

3.3.3.1 CC_AVAILABLE

In this state, the generic CCSS agent must wait for the `CallCompletionRequest` application or manager action to be executed. If one of these items occurs, then the core should be moved to the `CC_CALLER_REQUESTED` state if the request is legitimate. On an invalid request, the request is ignored and the state remains unchanged.

A state change from `CC_AVAILABLE` to `CC_CALLER_REQUESTED` does not match the original state diagram presented in this document; however, during the design of the CCSS generic agent, it was determined that this small change to the state machine flow was easier to follow and less esoteric than trying to bend the agent's behavior for the purpose of fitting the state machine flow.

If a call with a generic CCSS agent allocated finishes and the caller has not requested CCBS/CCNR, then the generic CCSS agent will request for the core to change to the `CC_CALLER_OFFERED` state.

3.3.3.2 CC_CALLER_OFFERED

The generic CCSS agent's job in this state is exactly the same as it is in the `CC_AVAILABLE` state, except it is also responsible for running an offer timer in this state. If a request has not been received when the offer timer expires, the core will be instructed to move to

the CC_FAILED state.

3.3.3.3 CC_CALLER_REQUESTED

During this state, the generic CCSS agent needs to be receptive to calls to the CallCompletionCancel application or manager action. If one of these occurs, then the generic CCSS agent will notify the core to change to the CC_FAILED state.

3.3.3.4 CC_ACTIVE

The generic CCSS agent's role is the same as in the CC_CALLER_REQUESTED state.

3.3.3.5 CC_CALLEE_READY

Upon reaching this state, the generic CCSS agent will query the device state of the calling party. If the calling party is not available (i.e. the calling party's state is not “not in use”), then the generic CCSS agent will request that the core change to the CC_CALLER_BUSY state.

If the calling party is available, then the generic CCSS agent will originate a call to the calling party. If the calling party does not respond to the originated call, the generic CCSS agent will request that the core change to the CC_FAILED state.

If the caller does answer the originated call, then the CCSS agent will run the callback_macro on the calling party's channel. When the macro completes, and assuming it does so successfully, the CCSS agent will place an outgoing call to the called party and request for the core to change state to CC_RECALLING. If the macro completes unsuccessfully, then the generic CCSS agent will request that the core move to the CC_FAILED state.

3.3.3.6 CC_CALLER_BUSY

During this state, the generic CCSS agent will subscribe to the device state of the calling party. When the calling party's device becomes “not in use,” the generic CCSS agent will request for the core to change to the CC_ACTIVE state.

3.3.3.7 CC_RECALLING

The generic CCSS agent has no specific roles during this state.

3.3.3.8 CC_COMPLETE

Once this state is reached, any unfreed memory should be freed.

3.3.3.9 CC_FAILED

If there is any memory to free, do it.

4. The CCSS Monitors

4.1 Overview

Monitoring a called party is a tricky concept. What happens if a call is placed to an extension which dials a SIP phone, an analog phone, and an ISDN phone? The analog phone will definitely need the support of a "generic" monitoring system since analog devices do not have native support for CCBS/CCNR. The SIP and ISDN phones may or may not be able to use native methods for reporting availability. In addition, some sort of multiplexer is needed in order to funnel the states of the three devices into one report to the core.

4.1.1 Configuration options

Below are descriptions of the configuration options which pertain to CCSS monitors.

*`cc_monitor_policy` – This option controls how and what type of CCSS monitor to use. The four valid options are:

- `never` – Never request CC services from this device
- `native` – If the peer has indicated that he supports CC services, then use protocol-specific messaging to request CC services. If the peer has not indicated that he supports protocol-specific CC services, then requesting CC services is not possible with this option set. This option is recommended for "trunk" device settings.
- `generic` – Whether or not the peer indicates that he supports native signaling for CC services, use a generic device monitor.
- `always` – If the peer indicates that he supports CC services at the protocol level, then use protocol-specific messaging to request CC services. If the peer does not, then a generic monitor will be used instead.

*`ccnr_available_timer` – This option describes the number of seconds to wait for the

called device to become available after CCNR services have been activated.
*ccbs_available_timer – This option describes the number of seconds to wait for the called device to become available after CCBS services have been activated.
*cc_max_monitors – The maximum number of monitor structures which may be created for this device. In other words, this number tells how many callers may request CC services for a specific device at one time.

4.1.2 Monitor structure

In order to facilitate potential complexity, there are two types of CCSS monitors used, extension monitors and device monitors. Monitors may be thought of as forming a tree structure, with the device monitors as leaves and extension monitors as roots. All CCSS monitoring begins with an extension monitor responsible for monitoring the extension that was originally dialed. This extension monitor will have, as children, device and possibly more extension monitors to monitor the devices and extensions dialed from within the initial extension. To clear things up a bit, here is a dialplan snippet and the resulting monitor tree.

```
[example]
exten => 50,1,Dial(SIP/2000&Local/100@example)
exten => 100,1,Dial(SIP/3000&mISDN/4000)
```

Consider that a caller dials 50@example. During the call, the dial application keeps track of all devices and extensions dialed during the call. If the call is unsuccessful, then the channel driver will decide if CCSS should be offered and will return this information to the dialing application. Monitor structures will be created once CCBS/CCNR has been requested by the caller. An extension monitor is first created for extension 50 since that was the original extension dialed. A device monitor for SIP/2000 and another extension monitor for 100@example are added as children to the originally-created extension monitor. Two device monitors for SIP/3000 and mISDN/4000 are added as children to the extension monitor for extension 100. A graphical representation of the tree formed can be seen in Figure 2.

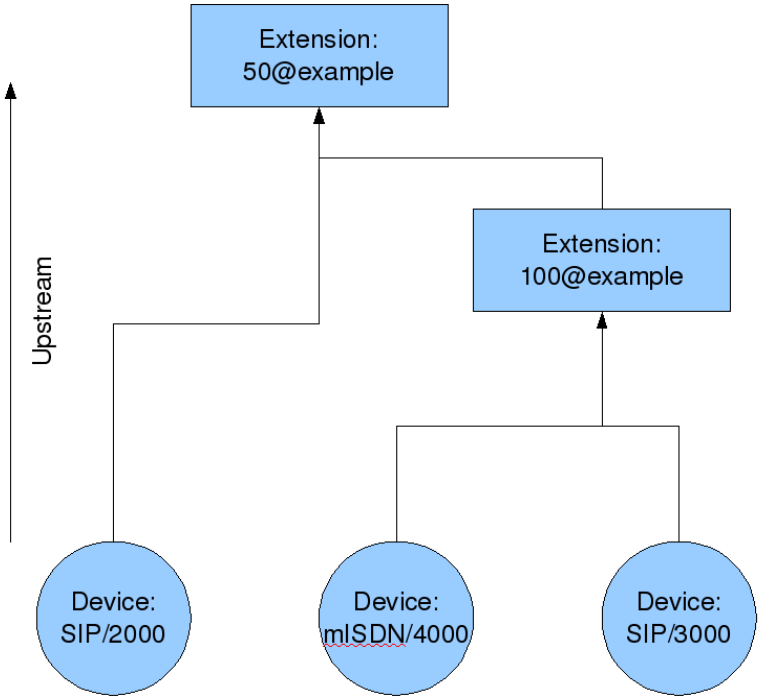


Figure 2

Extension monitors use the availability information given to them by their children and report this "upstream." In the case of the root of the tree, reporting upstream means reporting to the CCSS core. Extension monitors have complications since they have to report a single state of availability upstream, even though their underlying device monitors may report differing states of availability to them.

There are two types of device monitors, protocol-specific and generic. Protocol-specific device monitors report the availability of the device based on signaling received. They also are responsible for delivering any necessary outbound signaling in order to advertise the desire for notification of availability. Generic device monitors use the device state facility of Asterisk in

order to determine availability of the device. Device monitors will always have a parent extension monitor.

Something to consider is how to present conflicting reasons for unavailability to a caller. For instance, say that a caller were to dial an extension which then dialed two devices. One device was busy at the time the call was placed, and the other device never answered the incoming call. In the grand scheme of things, the calling party and the CCSS agent really don't care whether they are waiting on a busy subscriber or an unresponsive party. They just need to be notified when they can try calling back again. The core will use the hangup cause to determine whether to offer CCBS or CCNR to the calling party.

Similarly, it is important to determine how an extension monitor should behave when trying to report availability. If an extension monitor is monitoring two devices, and one signals that it has become available, is that enough for the extension monitor to report upstream that the called party is now available, or should the extension monitor wait for all underlying devices to report themselves as available before reporting its availability upstream? In this document, we specify that the availability of any single called party is enough to notify the calling party that he may attempt his CCBS or CCNR call back.

4.2 CCSS Queuing

Earlier, it was stated that the CCSS core is responsible for maintaining a queue of callers. This was not delved into very far because in order for the queuing mechanism to be understood, one must understand the tree of extension and device monitors that is created during a call.

First, let's consider why a straightforward FIFO queue in the core will not be sufficient. If a single queue were used in the core, the queue would have the same problem that `Queue()` in Asterisk had before `autofill` was implemented. That is, the callers beyond the head of the queue cannot be serviced even though it may be possible for them to be. They must wait until the first caller has been serviced before they may be addressed. A naïve attempt to fix this would be to implement a queue of callers per root extension monitor. The problem here is that there is the possibility of starvation or out-of-order servicing. Consider that two different callers may dial two different extensions. However, there is one common device that is dialed by both extensions.

As a result, two identical device monitors are created. If that device were to become available, the two device monitors would signal upstream that the device is available. It is unknown which of the root extension monitors would receive this signal first, meaning that there is no way to determine which of the two extension monitors the core would hear back from first. This race condition, under some really rotten circumstances, could even lead to starvation of a caller that should have been serviced long before others.

So how is the queue implemented? It's actually quite simple. The core assigns a number to each CCBS/CCNR request. This number is a monotonically-increasing integer, so if the requests were listed in numerical order, they also would be in chronological order. The number assigned is used as a means of providing a weight to links in the tree of monitors created. When viewed in concert, all active CCBS/CCNR requests form a weighted graph which illustrates how the monitors should behave.

Extension monitors and device monitors have a very strict set of rules to follow. Extension monitors follow the rule that if they receive an availability update from downstream, then, if they are going to indicate this availability upstream, they must do so upstream on the link with the same weight as the downstream link from which the update was received. Since an extension monitor may have multiple device monitors as children, it is not always necessary for the extension monitor to signal the availability of a device upstream since the availability of a "brother" device may have already been indicated upstream. Device monitors have two rules to follow. The first is that when the device they are monitoring becomes available, they must signal the availability on the lowest-weighted upstream link. The other rule they follow is that if an upstream link is destroyed, and the device they are monitoring is currently available, then they must signal upstream on the lowest-weighted upstream link remaining. Destruction of links is discussed in more detail below when discussing the monitor's role during the `CC_COMPLETE` and `CC_FAILED` states.

To better understand how this works, let's consider an example scenario with the following dialplan:

```
[example]

exten => 1000,1,Dial(SIP/1000)
exten => 2000,1,Dial(SIP/1000&SIP/2000)
exten => 3000,1,Dial(Local/2000@example&SIP/3000)
```

Caller C1 dials extension 1000. The monitor tree formed will look as it does in Figure 3. Later, Caller C2 dials extension 2000. The monitor tree formed for this call is represented in Figure 4. Then later, caller C3 dials extension 3000. The monitor tree formed for this call is represented in Figure 5. The combined weighted graph of all outstanding CCBS/CCNR requests can be seen in Figure 6.

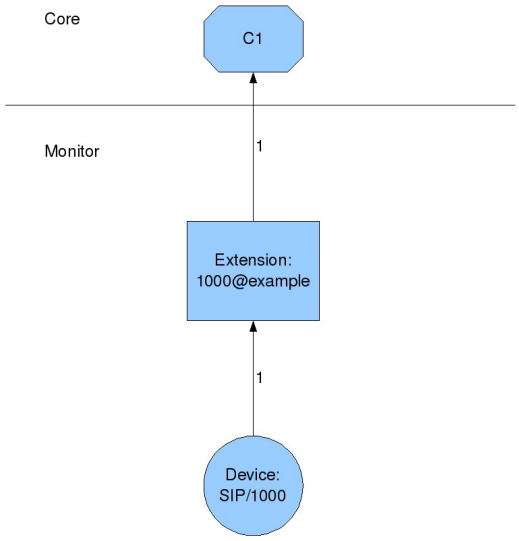


Figure 3

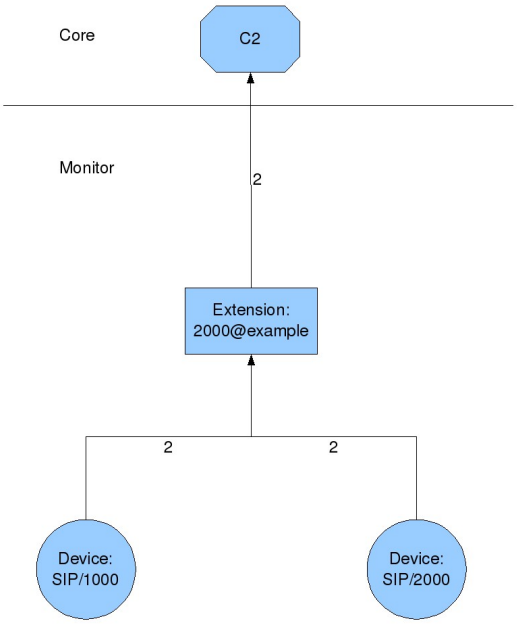


Figure 4

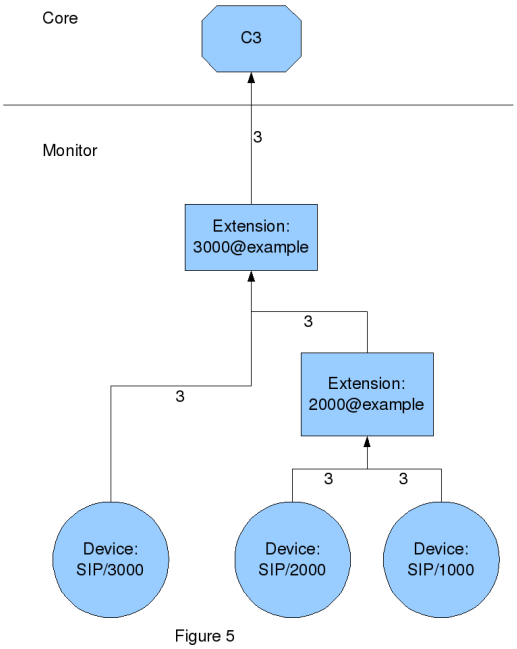


Figure 5

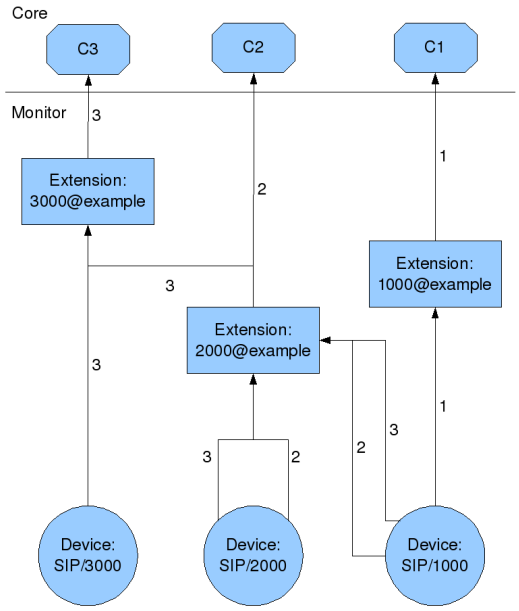


Figure 6

Let's consider some possible scenarios that arise in here. Consider first of all what happens if SIP/1000 were to become available. The device monitor watching the state of SIP/1000 has three upstream links. The rule states that the monitor should signal the availability on the lowest-weighted upstream link. In this case, this is the link with weight 1. This link leads to the extension monitor for extension 1000. The extension monitor follows its rule which means that it forwards the availability up the link with the same weight from which it was received. This means that the extension monitor will signal the availability up link 1 which leads to the core. From here, the core knows to notify C1's agent that the called party is available and can be called back.

Now let's look at what would happen if SIP/2000 were to become available instead of SIP/1000. In this case, the device monitor which watches SIP/2000 will signal the availability up its lowest-weighted link, which is link 2. Then the extension monitor for extension 2000 follows its rule by signaling the availability up the link with the same weight from which he received the signal, link 2. This leads to the core notifying C2's agent that he can attempt his call again. Note

here that C2 was able to cut in line in front of C1 because no devices that C1 was wishing to call became available before the devices that C2 was attempting to dial. It can be seen by following the same logic that if SIP/3000 were to become available first instead of either of the other devices, the result would be that C3 would be serviced first.

Let's take a closer look at a more complicated case. For this case, caller C1 never called in, but C2 and C3 did. The graph for this scenario can be seen in Figure 7. Now, consider that SIP/1000 becomes available and shortly after, SIP/2000 also becomes available. Following the rules for both, the eventual result is that C2 will attempt his CCBS/CCNR call back. Consider that SIP/1000 answers the call, and SIP/2000 remains available. Since the CCBS/CCNR call was successful for C2, the core can decommission the monitors set in place for C2. The result is that all the links with weight 1 are destroyed. When the link from the device monitor monitoring SIP/2000 and the extension monitor for extension 2000 is destroyed, then the device monitor will enact his second duty described. Since SIP/2000 is still available and an upstream link was destroyed, he must signal the availability of the device on the next-lowest-weighted link upstream. This means that the device monitor for SIP/2000 will signal on the link with weight 2 that SIP/2000 is available. This will result in the availability propagating upstream until the core signals the agent for caller C3 that he may attempt his call again. It can be seen from this example that callers attempting to dial a common device through different extensions will be serviced in chronological order and that there is as small a delay as possible between servicing multiple callers who are calling the same devices.

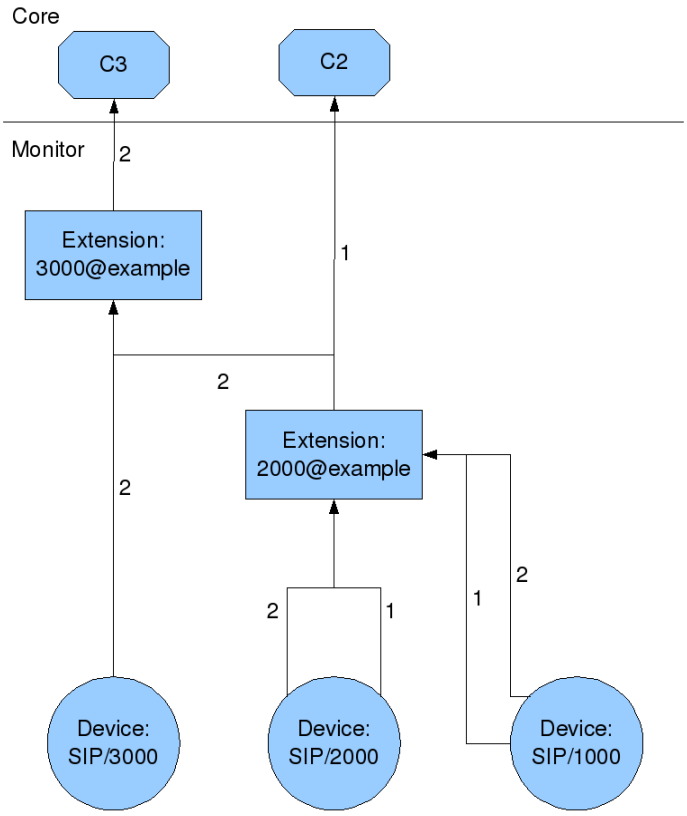


Figure 7

4.3 CCSS Monitor Suspension

A requirement of CCSS device monitors is that they provide the ability to suspend monitoring of a device. In the case of protocol-specific device monitors, the suspension is accomplished using some sort of signaling at the protocol level. For generic device monitors, they will either ignore device state updates of the device they are monitoring or they will temporarily unsubscribe to device state events for the device they are monitoring.

Suspension of monitoring also must be reflected in the weighted graph of CCSS monitors. For instance, if a device monitor detects that the device it is monitoring has become available, then its instinct will be to pass this change up its lowest-weighted link. However, what if this link leads to a caller which has had monitoring suspended? This problem can be avoided

by embedding whether the monitoring is suspended in the links themselves. As a result of this, the algorithm used by device monitors is slightly altered from what was presented earlier. Now, instead of reporting the device state change on its lowest-weighted link upstream, it now will report the change on the lowest-weighted link which is not currently suspended.

When does suspension of a monitor occur? If a caller has requested CCBS/CCNR and one of the devices being monitored becomes available, but the caller has now become unavailable for whatever reason, then monitoring of the called devices is suspended. This is accomplished by starting at the root extension monitor for the caller in question and marking all downstream links for the call as suspended. Device monitors have specific duties they must take on when their upstream links change suspension status.

When a device monitor is notified that one of its upstream links has become suspended, it should determine if the link which just became suspended was previously the lowest-weighted unsuspended link. If so, then if the device which the device monitor is monitoring is currently available, the device monitor should respond to this change of events by immediately advertising the availability of the device on the current lowest-weighted unsuspended link.

When a device monitor is notified that one of its upstream links has become unsuspended, it should determine if the newly-unsuspended link is the lowest-weighted upstream unsuspended link. If it is, and the device monitor has not yet notified any other upstream links of the device's availability, and the device being monitored is currently available, then the device monitor should immediately notify the upstream extension monitor that the device is available.

4.4 CCSS Monitor and the Core State Machine

The following are the CCSS monitor's duties during each state of the core state machine.

4.4.1 CC_AVAILABLE

During this state, the monitor will not yet have been created, so there are no duties to perform.

4.4.2 CC_CALLER_OFFERED

During this state, the monitor will not yet have been created, so there are no duties to perform.

4.4.3 CC_CALLER_REQUESTED

Typically when this state is reached, the monitor has not yet been created. The first step here is that the monitor tree for the call needs to be created and merged into the rest of the existing CCSS monitor weighted graph.

If it has not done so already, protocol-specific device monitors should send whatever protocol-specific signaling is necessary to start monitoring the called party. Protocol-specific device monitors should await acknowledgment of their CCSS request from downstream. When an acknowledgment is received, the protocol-specific device monitor should request for the core to change to the `CC_ACTIVE` state. If no acknowledgment is received, then the protocol-specific device monitor should request for the core to change to the `CC_FAILED` state.

Generic device monitors should subscribe to the device state of the device that they are to monitor. Since there is no acknowledgment to be received from downstream, generic device monitors should immediately request for the core to change to the `CC_ACTIVE` state.

4.4.4 CC_ACTIVE

When this state is entered, all device monitors should start their configured `cc_available_timer`s if not already running. It is possible that upon entering this state, the called party's monitor is currently suspended. If that is the case, then the monitor should unsuspend its suspended links for the call. Device monitors should be receptive to notifications of availability. If a device should become available, the device monitor should notify the upstream extension monitor using the rules outlined in section 4.2. If an extension monitor receives a notification that one of its child device monitors has reported availability, then the extension monitor should request for the core to change to the `CC_CALLEE_READY` state.

It is possible that upon entering this state, the called party's monitor is currently suspended. If that is the case, then upon entering this state, the first thing that should be done is to unsuspend the links in the monitor.

4.4.5 CC_CALLEE_READY

Monitors should continue to watch the status of the called party during this state. While there is no need for device monitors to report availability upstream, it is still worthwhile to cache

availability status.

4.4.6 CC_CALLER_BUSY

During this state, all monitoring for this transaction shall suspend. For details about suspension of monitors, see section 4.3.

4.4.7 CC_RECALLING

Monitors have no specific duties during this state.

4.4.8 CC_COMPLETE

Starting with the root extension monitor, begin moving down the tree representing the current call, destroying all links encountered. If at any point a monitor, whether it be extension or device, becomes orphaned (i.e. there are no more remaining upstream links) then the monitor will be destroyed. During this phase, if a link to a device monitor is destroyed, the device being monitored is available, and there are any upstream links remaining, then the device monitor must signal availability upstream on the lowest-weighted, unsuspended link remaining. In addition, all device monitors should kill the associated running available timer.

4.4.9 CC_FAILED

CCSS monitors take the same action in this state as they do in the `CC_COMPLETE` state.

5. Examples

Presented here are three examples. The first example will make use of only one Asterisk box and use only the generic CCSS agent and a generic CCSS device monitor. The second example will demonstrate a call across multiple Asterisk boxes to multiple destinations over multiple protocols. The second example will also demonstrate a situation which does not go exactly as planned and shows how the architecture copes with these problems. The third example also involves multiple Asterisk servers, but its focus is much less on the internal workings of the CCSS system and more on how the dialplan and configuration affect the situation.

5.1 Example 1

For this example, refer to the dialplan presented in section 4.2. The dialplan presented is limited to showing what devices are dialed; it does not show all dialplan steps necessary to enable CCSS for a call. For a more complete example dialplan, see section 5.3. This example is meant to illustrate the creation of a monitor graph and the process of the core state machine on a single Asterisk box.

Alice is trying to reach her coworker, Bob, at extension 1000 to discuss some upcoming meetings the two will be attending. Unfortunately, when Alice called, Bob was on the phone with Carol discussing finances. When Alice was dialing Bob, app_dial kept track of the extensions and devices which were dialed during the call. app_dial determined that extension 1000@example was dialed to reach device SIP/1000, which is Bob's phone. The Dial application presents this information to the CCSS core. The CCSS core determines that the called party is a directly-connected phone and thus can be monitored using a generic device monitor. Since the core has determined that call completion services are available for this call, it starts the state machine in the `CC_AVAILABLE` state and creates a generic CCSS agent. Upon hangup, the core changes the CCSS state to `CC_OFFERED`, and notifies the generic agent of the change. The generic agent arms its offer timer in case Alice does not promptly request CCBS.

Alice, being familiar with the corporate phone system, knows that she may dial *02 in order to request CCBS for the call she previously attempted. Since she would like to speak to Bob again right away, she dials *02. Once Alice has requested CCBS, she hangs up. The generic CCSS agent requests that the core move to the `CC_CALLER_REQUESTED` state and disarms its previously running offer timer. The core receives this request and changes to the `CC_CALLER_REQUESTED` state.

At this point, the core actually allocates resources for monitoring Bob. The core creates a monitor tree exactly like the one in Figure 3. The core then alerts both the agent and monitor components that the state has changed to `CC_CALLER_REQUESTED`. The newly-created generic device monitor starts the configured `ccbs_available_timer`. Since the device monitor is generic, it immediately signals to its parent extension monitor to request that the core change to the `CC_ACTIVE` state. The core complies with the request and changes states.

Since the device monitor created by the core is a generic device monitor, the device monitor subscribes to device state changes for SIP/1000. After about 15 minutes, Bob's conversation finally finishes and he hangs up his phone. The generic device monitor is notified that the device state of Bob's phone has become “not in use.” The device monitor then finds the lowest-weighted upstream link which is currently not suspended. In this case, this is the only upstream link, which leads to the extension monitor for extension 1000@example. The device monitor reports upstream that Bob is now available. The extension monitor, having received such notification, and seeing that the current state for this CCBS call is `CC_ACTIVE`, then reports to the core to change states to the `CC_CALLEE_READY` state.

The core changes the state and alerts the agent and monitor of the change. The generic agent, seeing that the state has changed to `CC_CALLEE_READY`, now checks the device state of Alice's phone. She still is available, so the generic agent originates a call to Alice's phone. When she answers, the generic agent runs the configured callback_macro. The macro tells Alice that she is being called because of her CCBS request. The macro tells her to wait while Asterisk tries to connect her to Bob. The callback_macro completes and the generic CCSS agent places an outgoing call to Bob's extension. After placing the call, the CCSS agent requests that the core change to the `CC_RECALLING` state. The core makes the change as requested. When Bob's phone sends a ringing message to the channel driver calling Bob's phone, the channel driver requests for the core to change states to `CC_COMPLETE`. Soon, Bob answers the phone.

The core changes to the `CC_COMPLETE` state and alerts the agent and monitor of the change. The core erases the record for the previous failed call from Alice to Bob. The agent and monitor free any allocated resources for this call (this includes the generic monitor killing the available timer) and then the core destroys the agent and monitor.

5.2 Example 2

Roger has a problematic new computer. Roger has done all the diagnosis he can on his new computer and has decided to call the manufacturer's technical support line in order to hopefully get them to repair or replace it. Roger starts the process by picking up his SIP phone and dialing the number listed on the manufacturer's web site. For this specific call, the arrangement of servers over which the call travels can be seen in Figure 8.

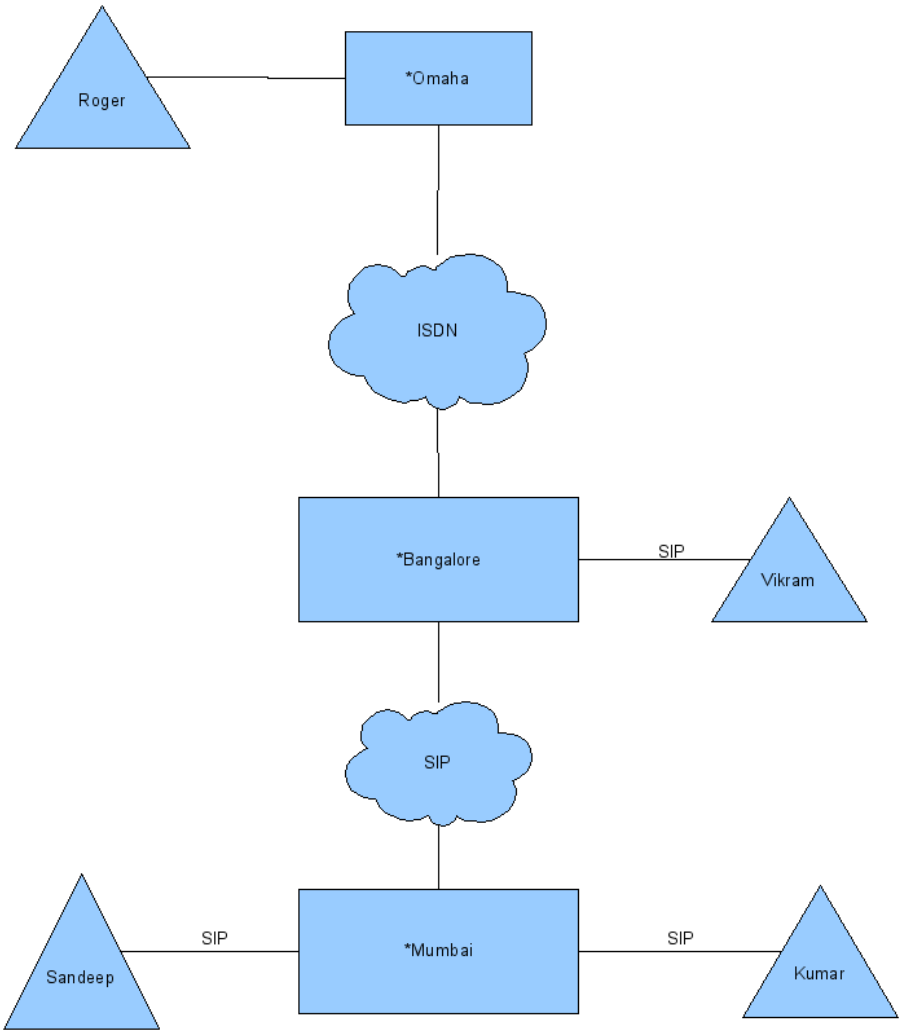


Figure 8

The call travels from Roger's phone to his Asterisk box in Omaha and then over ISDN to an Asterisk box in Bangalore. From there, the call is forked out to Vikram, who is at the call center in Bangalore, but then also to an Asterisk server in Mumbai, where the call reaches

Sandeep and Kumar's phones. Vikram and Kumar are currently on the phone with other customers. Sandeep has stepped away from his desk and is unable to answer his ringing phone.

Focusing on *Mumbai, the Dial application determines that the call has not been successfully answered and queries the CCSS core to determine if call completion services may be available for this call. It then goes through its checks to see if CCBS/CCNR is available for the call. Neither Sandeep nor Kumar's phone supports CCBS/CCNR natively, but since they are both connected directly to *Mumbai, they are capable of CCBS/CCNR support. The incoming SIP link from *Bangalore is configured to support CC services over SIP. As a result, *Mumbai's CCSS core creates a SIP protocol-specific agent. *Mumbai sends a SIP response to *Bangalore which contains an offer for CCNR.

At this point, *Bangalore has determined that the dial attempt has failed to all devices. The core in *Bangalore now undergoes the same process that the CCSS core in *Mumbai did. It determines that both called devices are CCBS/CCNR-capable and that since the ISDN link on the caller side is configured to be CCBS/CCNR capable at the protocol level, CCBS/CCNR is available for the call. *Bangalore creates an agent for the calling side then sends a response through the ISDN network advertising CCBS/CCNR availability for the call.

Finally, *Omaha goes through the same process. It determines that the called device supports CCBS/CCNR natively and that the calling party can be offered CCBS/CCNR using the generic agent. At this point, all CCSS agents on all Asterisk servers involved have been created and are in place. The CCSS cores on *Mumbai and *Bangalore are in the `CC_OFFERED` state since they have offered CC services upstream, but *Omaha is still in the `CC_AVAILABLE` state since no offer has been given to Roger.

*Omaha is configured such that a prompt tells Roger that he may dial *02 after the call has completed to request CCNR. At this point, the CCSS cores on all Asterisk boxes in the path have kept in memory necessary information so that when CCNR is requested, they can verify that the request is valid. Since the call has completed on all Asterisk boxes, all CCSS cores should be in the `CC_CALLER_OFFERED` state, meaning that the `cc_offer_timer`s on all Asterisk boxes are currently armed and running.

Roger decides to request CCNR and dials *02. The generic CCSS core on *Omaha determines that this is a legitimate request for CCNR and alerts the core to change states to `CC_CALLER_REQUESTED`. The agent then stops the `cc_offer_timer` which had been previously running. The core honors the request and creates a CCSS monitor based on the saved called party data. When the CCSS monitor on *Omaha is created, it will send out a CCNR request over the ISDN network to *Bangalore.

When *Bangalore receives the CCNR request, the ISDN CCSS agent will stop its running `cc_offer_timer` and request that the CCSS core on *Bangalore changes its state to `CC_CALLER_REQUESTED`. The core there will proceed to change state and create an appropriate monitor tree structure. Since the device monitor for Vikram is a generic device monitor, no specific indications need to be sent there; the device monitor will just subscribe to the device state of Vikram's phone. However, the SIP link between *Bangalore and *Mumbai uses a SIP device monitor, and so a SIP CCNR request is sent over the link.

The SIP CCSS agent in *Mumbai receives the SIP CCNR subscription request, requests that the core change states to `CC_CALLER_REQUESTED`, and stops the running `cc_offer_timer`. The core goes through the same steps as *Omaha and *Bangalore. The state is changed and monitor structures are created for Kumar and Sandeep's phones. Since both the device monitor for Kumar and the device monitor for Sandeep are generic device monitors, there is no special signaling to be sent to them; the monitors will just subscribe to the device state of the two phones.

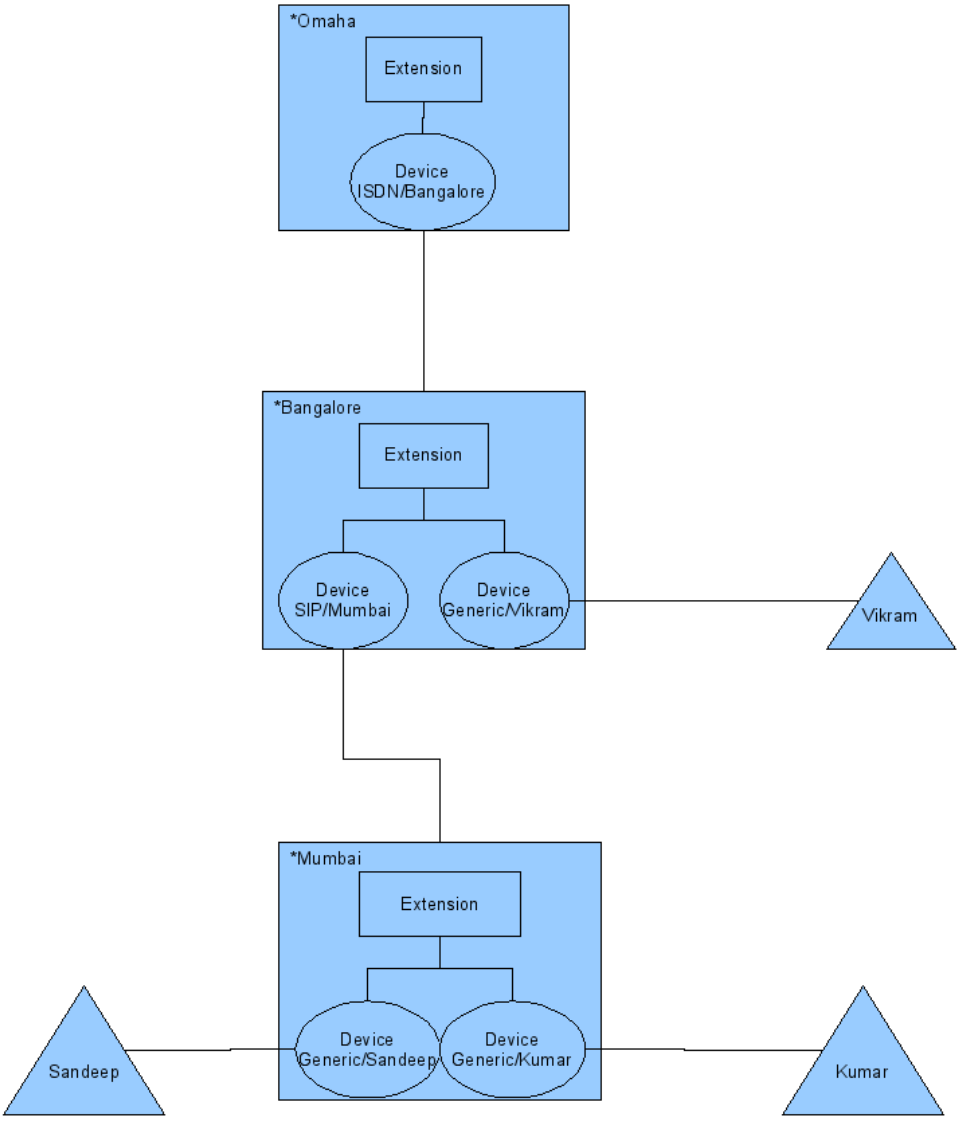


Figure 9

Since both device monitors in *Mumbai are generic, there is no need to wait for an acknowledgment of the CCSS subscription requests. Instead, the monitors immediately signal to

their parent extension monitor that it may request that the core change states to `CC_ACTIVE`. After the extension monitor has heard from both of the child device monitors, it requests that the core change its state to `CC_ACTIVE`. The core complies, changes state and alerts the CCSS agent and device monitors of the change. The device monitor for Kumar's phone begins running its `ccbs_available timer`, and the the device monitor for Sandeep's phone begins running its `ccnr_available_timer`. The SIP CCSS agent in *Mumbai sends a positive acknowledgment of the received SIP CCNR subscription it had received earlier from *Bangalore.

The SIP device monitor in *Bangalore receives the acknowledgment of the subscription request and passes this information up to its parent extension monitor. The extension monitor, having received positive acknowledgments from both the generic device monitor for Vikram's phone and the SIP device monitor for *Mumbai then requests for the core to change to the `CC_ACTIVE` state. The core makes the change and notifies the agent and monitor. The ISDN CCSS agent receives the notification and sends an acknowledgment of the CCNR subscription to *Omaha. The device monitors in *Mumbai, in response to the change to `CC_ACTIVE`, begin running their respective available_timers.

The ISDN device monitor in *Omaha receives the CCNR subscription acknowledgment from *Bangalore and goes through the same process as the other Asterisk servers. Since the CCSS agent in use in *Omaha is a generic agent, it does not have to send any sort of acknowledgment upstream. The ISDN device monitor starts its `ccnr_available_timer`.

After a few minutes, before any of the availability timers have expired, Kumar finishes his previous call. The device monitor in *Mumbai for Kumar reports the device state change upstream to the extension monitor for the call. The extension monitor alerts the core of the called party's availability and requests that the core change states to `CC_CALLEE_READY`. The core then changes states. The SIP CCSS agent is alerted of the change and sends a SIP message over the SIP link to *Bangalore to notify *Bangalore of Kumar's availability.

The SIP device monitor in *Bangalore receives the notification and notifies its parent extension monitor. The extension monitor requests for the core to change state to

CC_CALLEE_READY. The core does so. The ISDN CCSS agent on *Bangalore then sends an ISDN message indicating the called party's availability to *Omaha. The ISDN device monitor on *Omaha notifies its parent extension monitor, who then requests that the core on *Omaha change to CC_CALLEE_READY. The core does so.

Now the generic CCSS agent in *Omaha checks the device state of Roger's phone. It's in use right now, because Roger has received a call from a coworker asking when Roger is going to call tech support to get his computer fixed. As a result, the generic CCSS agent in *Omaha requests for the core to change its state to CC_CALLER_BUSY. The core changes its state as told. The ISDN device monitor in *Omaha then sends a request to suspend monitoring of the called party to *Bangalore.

The ISDN agent on *Bangalore receives the request and asks the core to change to CC_CALLER_BUSY. The core changes state as asked. The generic device monitor for Vikram's phone does not take any specific action as a result, but it knows that it does not need to report device state changes upstream. The SIP device monitor, though, sends a SIP monitor suspension request to *Mumbai. The SIP agent in *Mumbai receives the request and asks that the core change to the CC_CALLER_BUSY state. The core complies. The two generic device monitors, upon hearing of the change, take no specific action but know not to notify upstream entities of device state changes on Kumar or Sandeep's phones.

Time passes. Eventually, Roger finishes talking with his coworker. Roger hangs up his phone, which causes the generic CCSS agent in *Omaha to request that the core change to the CC_ACTIVE state. The core makes the change and notifies the monitors. The ISDN device monitor on *Omaha then sends a signal to *Bangalore to unsuspend the suspended monitoring. The ISDN agent on *Bangalore gets this message and requests that its core change to CC_ACTIVE. The core complies with the request. The generic device monitor for Vikram's phone does nothing here since Vikram is still unavailable. The SIP device monitor sends *Mumbai a SIP request to stop suspension of monitoring. The SIP agent on *Mumbai receives the request and requests that its core change to CC_ACTIVE. The core does so and notifies the monitors.

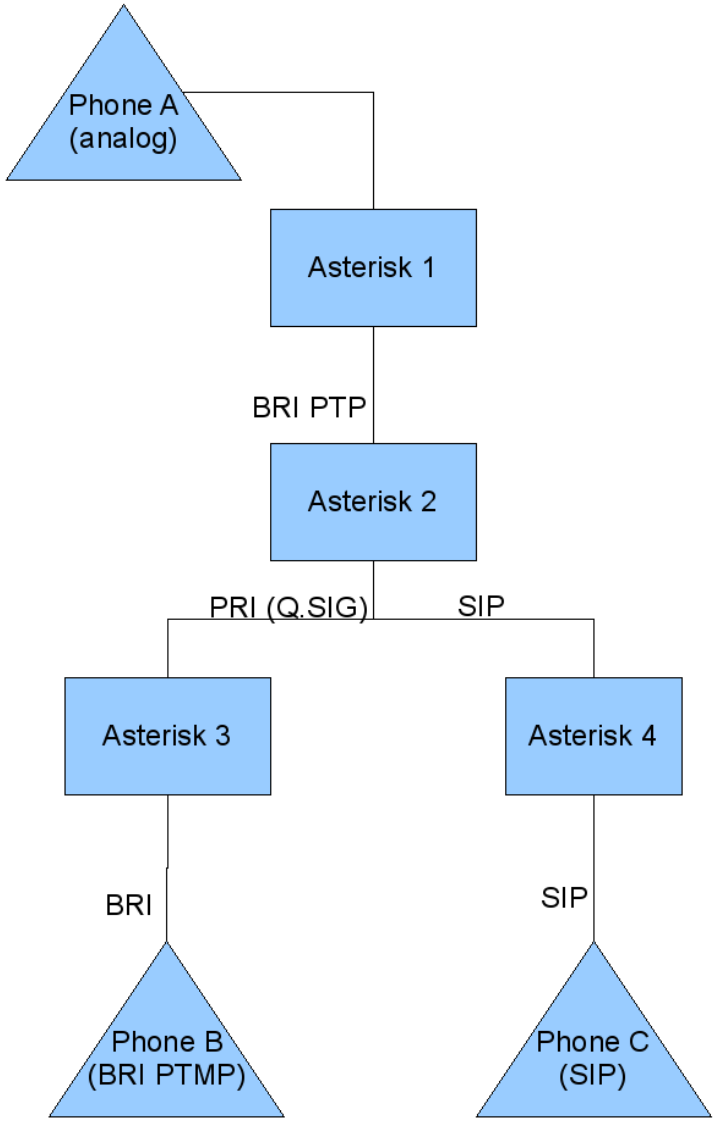
The generic device monitor for Kumar's phone receives the notification to stop suspending monitoring, and since Kumar is available, the monitor immediately requests that the core change to `CC_CALLEE_READY`. The progression of state changes that happened previously happens again all the way up to *Omaha. This time, the generic agent for Roger's phone finds that Roger is available.

The CCSS agent originates a call to Roger's phone. There is no callback_macro set on *Omaha, so nothing extra occurs when Roger answers. Upon answering the call, the agent then makes an outbound call to the extension that Roger originally dialed and requests that the core change to the `CC_RECALLING` state. This will result in the call going to the same destinations that it originally went to. As each agent on each server receives the new call, it will place an outbound call to each of the appropriate destinations and request for the core to change to `CC_RECALLING`.

Once *Mumbai receives a progress indication from either of Sandeep or Kumar's phones, it requests for the core to change to the `CC_COMPLETE` state. Upon doing so, the monitor structure for the call on *Mumbai is torn down, and the core frees any resources for this CC call. The ringing indication is sent to *Bangalore. Once one of the two device monitors in *Bangalore have received progress indications, the extension monitor in *Bangalore requests for the core to change to `CC_COMPLETE`. The monitor structures are destroyed, and a ringing indication is sent to *Omaha. Reception of the ringing indication causes the state to change to `CC_COMPLETE` in *Omaha and for all resources for the CCNR transaction to be freed. Kumar answers his ringing phone this time and the call is connected.

5.3 Example 3

For this example, the focus will be less on the state machine transactions and signaling between systems and more on dialplan and configurations that exist on machines. For this scenario, the setup is as shown in Figure 10. A person places a call from phone 'A' to recipients at phones 'B' and 'C.'


Phone A
(analog)
Asterisk 1
BRI PTP
Asterisk 2
PRI (Q.SIG)
SIP
Asterisk 3
Asterisk 4
BRI
SIP
Phone B
(BRI PTMP)
Phone C
(SIP)

Asterisk box 1 has the following dialplan:

```
[local_phones]
exten=>4000,1,Set(DESTS=IF(EXISTS(${CC_INTERFACES})?${CC_INTERFACES}:DAHDI/g1/${EXTEN}))
exten=>4000,n,Dial(${DESTS})
exten=>4000,n,GotoIf([${DIALSTATUS}=BUSY]?busy:no_response)
exten=>4000,n(busy),ExecIf(EXISTS(${CC_AVAIL})?Macro(cc_offer,USER_BUSY))
exten=>4000,n(no_response),ExecIf(EXISTS(${CC_AVAIL})?Macro(cc_offer,NO_ANSWER))
exten=>4000,n,Hangup
[macro-cc_offer]
exten=>s,1,Read(CC_REQUEST,IF($[${ARG1}=USER_BUSY]?ccbs-offer:ccnr-offer),1,n)
exten=>s,n,ExecIf($[${CC_REQUEST}=1]?CallCompletionRequest())
```

In addition, the chan_dahdi.conf on this machine looks like the following:

```
[channels]
cc_offer_timer=45
ccbs_available_timer=2700
ccnr_available_timer=6300
cc_agent_policy=generic
cc_monitor_policy=always
signaling=fxo_ks
channel=>1 ;Caller A. analog phone
ccbs_available_timer=3600
ccnr_available_timer=7200
cc_offer_timer=70
cc_agent_policy=native
cc_monitor_policy=native
signaling=bri_cpe
```

```
switchtype=euroisdn
channel=>8; Link from *1 to *2; BRI PTP
```

Asterisk box 2 has the following dialplan:

```
[from_ast1]
exten=>4000,1,Set(DESTS=IF(EXISTS(${CC_INTERFACES})?$
{CC_INTERFACES}:DAHDI/g1/${EXTEN}&SIP/${EXTEN}@peer4)
exten=>4000,n,Dial(${DESTS})
exten=>4000,n,GotoIf($[${DIALSTATUS}=BUSY]?busy:no_response)
exten=>4000,n(busy),Hangup(USER_BUSY,${CC_AVAIL})
exten=>4000,n(no_response),Hangup(NO_ANSWER,${CC_AVAIL})
```

Asterisk box 2 has the following chan_dahdi.conf file.

```
[general]
cc_agent_policy=native
cc_monitor_policy=native
ccbs_available_timer=3600
ccnr_available_timer=7200
cc_offer_timer=70
cc_max_agents=5
cc_max_monitors=5
```

The specific channels are not shown here because they are irrelevant for this example. By placing the configuration in the general section, the cc parameters are set for all channels in the file. The sip.conf file on Asterisk box 2 has the same settings in the general section.

Asterisk box 3 has the following dialplan:

```
[from_ast2]
exten=>4000,1,Set(DESTS=IF(EXISTS(${CC_INTERFACES})?$
```

```
{CC_INTERFACES}:DAHDI/5)
exten=>4000,n,Dial(${DESTS})
exten=>4000,n,GotoIf($[${DIALSTATUS}=BUSY]?busy:no_response)
exten=>4000,n(busy),Hangup(USER_BUSY,${CC_AVAIL})
exten=>4000,n(no_response),Hangup(NO_ANSWER,${CC_AVAIL})
```

Asterisk box 3 has the following for chan_dahdi.conf:

```
[channels]
cc_agent_policy=native
cc_monitor_policy=native
ccbs_available_timer=3600
ccnr_available_timer=7200
cc_offer_timer=70
cc_max_agents=5
cc_max_monitors=5
signaling = pri_cpe
switchtype = qsiq
channel => 2; Q.SIG link from *2 to *3
cc_agent_policy=generic
cc_monitor_policy=always
ccbs_available_timer=2700
ccnr_available_timer=6300
cc_offer_timer=45
cc_max_agents=5
cc_max_monitors=5
signaling=bri_net_ptmp
channel => 5; BRI PTMP link from *3 to phone B
```

Asterisk box 4 has the following dialplan:

```
[from_ast2]
```

```
exten=>4000,1,Set(DESTS=IF(EXISTS(${CC_INTERFACES})?${CC_INTERFACES}:SIP/${EXTEN})
exten=>4000,n,Dial(${DESTS})
exten=>4000,n,GotoIf($[${DIALSTATUS}=BUSY]?busy:no_response)
exten=>4000,n(busy),Hangup(USER_BUSY,${CC_AVAIL})
exten=>4000,n(no_response),Hangup(NO_ANSWER,${CC_AVAIL})
```

Asterisk box 4 has the following sip.conf:

```
[ast_2]
cc_agent_policy=native
cc_monitor_policy=native
ccbs_available_timer=3600
ccnr_available_timer=7200
cc_offer_timer=70
cc_max_agents=5
cc_max_monitors=5
[peer4]
cc_agent_policy=generic
cc_monitor_policy=generic
ccbs_available_timer=2700
ccnr_available_timer=6300
cc_offer_timer=45
cc_max_agents=2
cc_max_monitors=2
```

There are a few key concepts to note in this setup. First, notice that the channel configurations for all phones directly connected to an Asterisk server have shorter `cc_offer_timer`, `ccnr_available_timer`, and `ccbs_available_timer` values than any of the other channels. This is because it is most logical to have the servers nearest to the phones be the ones to trigger a timeout condition. While it is not required to have a setup like

this, it is recommended. Second, notice that the `cc_agent_policy` and `cc_monitor_policy` is always set to `native` on all channel configurations which link between Asterisk servers. For such connections, `native` and `never` are the only choices which will have well-defined behavior. This is because `always` will attempt to use a generic agent or monitor if protocol-specific ones cannot be used, and `generic` will always attempt to use a generic agent or monitor. Generic agents and monitors only work well for links between Asterisk and a phone.

Notice also that the dialplan on Asterisk servers 2, 3, and 4, are very similar. Asterisk server 1 would have a similar dialplan, too, except that Asterisk server 1 requires special handling in order to offer CCBS/CCNR to the caller since a generic agent must be used to represent analog phone A. If phone A were a phone which supported CCBS/CCNR at the protocol level, then the administrator could set the `cc_agent_policy` for phone A to "`native`" and use the same dialplan structure used in the other Asterisk servers.

The caller at phone A initially attempts to call the recipients at phones B and C. The calls reach the two phones, but the person at phone C is currently on the phone with someone else and the person at phone B is currently unable to answer the call. The `Dial` applications on Asterisk 3 and 4 both determine that CC services are available for the current call and thus set the `CC_AVAIL` variable to be non-empty. The `Hangup()` applications each set the second parameter to be non-empty as a result, meaning that the hangup messages sent have appropriate CC offers in them. Sending the CC offers begins the `cc_offer_timer`s running on servers 3 and 4.

Asterisk server 2 receives the hangup messages from servers 3 and 4. Since the Dial application receives the "no answer" hangup from server 3 last, the `DIALSTATUS` for the call will be set to "no answer." This means that the n(no_response) priority will be run in the dialplan, which means that the hangup message sent to server 1 will contain an offer for CCNR. Sending the hangup message starts the `cc_offer_timer` for server 2.

Asterisk server 1 receives the hangup message from server 2. Since the hangup contains a CC offer, the Dial application sets the `CC_AVAIL` variable to be non-empty. Further, the

`DIALSTATUS` is set to `NO_ANSWER`. The macro-cc_offer is run, prompting the user that he may press '1' in order to request CCNR. The user presses 1, leading to the `CallCompletionRequest` application being run. Server 1 has enough information stored for the call to know that CCNR is the appropriate request that should be made to server 2. Server 1 sends a CCNR request to Server 2.

Server 2 receives the CCNR request and is able to associate the request with the earlier failed call that it has saved. Server 2 kills the `cc_offer_timer` that had been running. Server 2 sends a CCNR request to server 3 and a CCBS request to server 4. Asterisk servers 3 and 4 receive their CCNR and CCBS requests and react mostly in the same way that server 2 does. The difference is in the method that CC monitoring is set up for phones B and C. Server 3 has saved as part of the information for the previously failed call that the phone it is communicating with is incapable of receiving CC requests, so a generic monitoring system will be used instead. Similarly, server 4 sets up a generic monitor for phone C, but it does so not because phone C is incapable of understanding CCBS messages but because the configuration parameter for phone C's `cc_monitor_policy` is set to `generic`.

Servers 3 and 4 each send appropriate messages to accept the CC requests from server 2. Upon doing so, the `ccnr_available_timer` is started for the link between server 3 and phone B, and the `ccbs_available_timer` is started for the link between server 4 and phone C. Similarly, server 2 sends a message to server 1 accepting its request. Server 2 starts the `ccbs_available_timer` for the link between server 2 and server 4 and the `ccnr_available_timer` for the link between server 2 and 3. When server 1 receives the acceptance message of its CCNR request, it begins the `ccnr_available_timer` for the link between server 1 and 2. At this point, the available timers are running on each of the servers. After a short time, the person at phone B places a quick call to someone and hangs up. At this point, the generic device monitor on server 3 detects the device state change of the phone and sends an appropriate notification to server 2. Server 2 receives this notification and sends an appropriate notification to server 1.

Server 1, upon receiving the notification, will originate a call to phone A. The person at

phone A answers the phone. When the phone is answered, server 1 sends an outgoing call to server 2, destroys the monitor structure set up for the link between itself and server 2 for this call, and stops the `ccnr_available_timer` from running. Server 1 also saves the information for this call just in case it should also result in a busy or no response situation.

Server 2 receives the incoming call and associates it with the CC requests it has with servers 3 and 4. Server 2 erases his monitor structure for the CC request, places outgoing calls to servers 3 and 4, stops the two running `ccnr_available_timer` and `ccbs_available timers`, and saves information for this call in case a busy or no response situation occurs. Servers 3 and 4 behave similarly, placing calls to phones B and C, respectively.

This time, phone C is still busy, but the person at phone B is able to answer. The successful answer is sent to server 3. Server 3 sends an answer to server 2. Server 2 will send a cancellation to server 4. Server 2 then sends a successful answer to server 1. Server 1 receives the answer and bridges the call.