M2 Cryptography and Authentication Framework

This document descibes the application of cryptography in the M2 network to provide secure communications between the service providers and consumers on the network.

There are three types of entities involved in the establishment of encrypted communication – the client, which connects to the network to consume services; the component, which provides those services; and the authenticator, which stores identity validation information (public keys, passwords, etc). Both the client and component first establish a secure channel to the authenticator before leveraging off this to setup secure communications between each other without needing to re-prove their identities.

The three types of channel setup that occur are:

- Client to authenticator, password based authentication, session RSA key generation for federated authentication
- Component to authenticator, encrypted channel, optionally authenticated using the component's private key
- Client to component, authenticated using the client's session RSA keypair

The following sections detail these channel setup processes in detail.

Glossary

- *CSPRNG* – Cryptographically Secure Pseudo Random Number Generator. A programmatic source of random numbers satisfying the “next bit” test – that there is no (computationally feasible) means of predicting the next bit with better than 50% accuracy, even given the sequence generated so far.
- *Authenticator* – The central component on the M2 network that manages identity information and provides the means to bootstrap the cryptographic trust network.
- *Client* – A software entity that connects to the M2 network to consume services made available through it.
- *Component* – A software entity that connects to the M2 network to make available services.
- *Junkmail* – A server initiated asynchronous update channel that can have multiple downstream subscribers, and for which disconnection notifications are send to the endpoints when broken.
- *m2_node* – The software component that provides message transport and routing.
- *req* – A message type indicating a service routed request.
- *rsj_req* – A message type indicating a request routed over a previously established junkmail channel, from the client end (reverse server junkmail).
- *ack* – A message type indicating a successful response to a request (acknowledgement).
- *nack* – A message type indicating an unsuccessful response to a request (negative acknowledgement).
- *pr_jm* – A message type indicating a junkmail channel setup, sent in response to a

req, before the ack or nack. Only the client that issued the request receives the pr_jm message (private junkmail).

- *jm* – A message type indicating a junkmail update on a previously established channel. All clients subscribed to the channel receive the update (junkmail).
- *jm_can* – A message type indicating that a channel has been cancelled, sent in the downstream direction (to the clients) from the break (junkmail cancel).
- *jm_disconnect* – A message type indicating that a channel has been cancelled, sent in the upstream direction (to the component) from the break (junkmail disconnect).
- *RSA* – An public-key cryptography algorithm developed by Ron Rivest, Adi Shamir and Leonard Aldeman at MIT. Security is based on the mathematical hardness of the integer factorization problem.
- *Keypair* – A matching public and private key used by public-key cryptosystem in which plaintext encrypted by one half of the key is only decryptable by the other half.
- *Blowfish* – A symmetric key cipher (the same key is used to encrypt and decrypt), developed by Bruce Scheier, which is fast, can use large keysizes, and is widely considered strong in full 16 round form (blowfish is a feistel cipher).

Client / Component – Authenticator established setup

This is the process by which the client or component bootstraps an encrypted channel to the authenticator, using the authenticator's public key, which it obtained either as part of the software distribution of the client / component, or through another trusted sideband channel. This procedure is common to both the client and component, the following description where it refers to the client includes the component too.

The process is initiated by the client in response to receiving a notification that the authenticator service is available from the m2 network. A cookie is generated (a random number generated from a CSPRNG) and is encrypted using the authenticator's public RSA key. The session key (a blowfish key generated at client startup) is separately encrypted with the authenticator's public key and is sent together with the encrypted cookie to the authenticator.

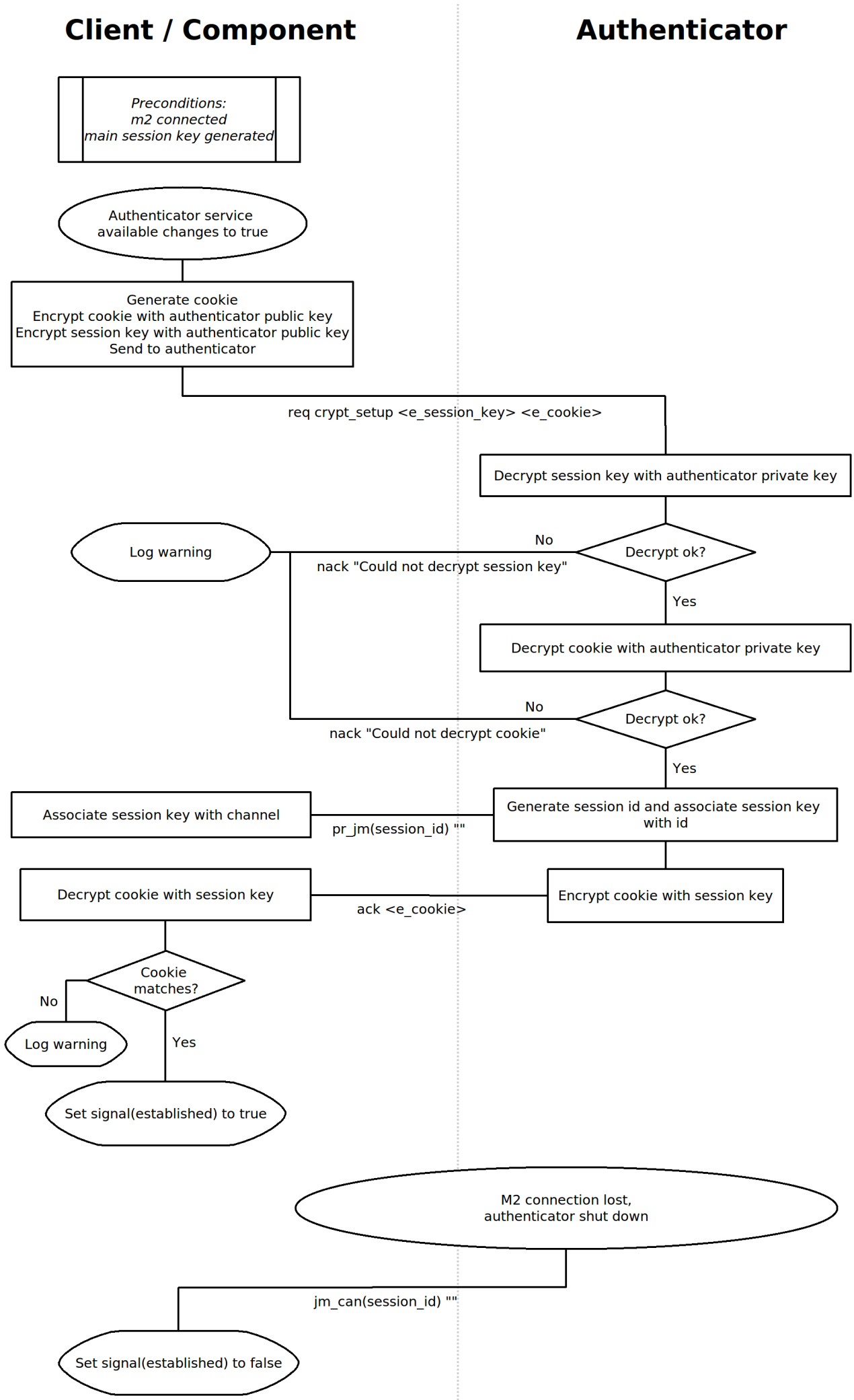
The authenticator attempts to decrypt both these using it's private key. If either decrypt fails, a message is sent to the client indicating failure and the process ends. If both were successful, a session ID is generated and associated with the client-supplied blowfish session key.

A junkmail channel (unique to this client instance) identified by this session ID is set up to the client. Upon receiving this junkmail setup message, the client records the channel ID and associates it with it's session key. All messages transmitted over this channel, from either the authenticator or client, are encrypted using this key.

The initial request is then acknowledged, with the payload being the cookie, now encrypted with the session key. Receiving this, the client decrypts it with the session key and compares it with the cookie it generated and sent to the authenticator in the first step. If they match, then the authenticator must be in possession of the private key matching the public key the client has stored (and trusts), and the session key must have been transferred intact and without possibility of interception.

At this stage the secure communication channel between the client and authenticator is set up, the client has verified the identity of the authenticator, but the authenticator has not yet verified the client's identity. The signal “established” is set to true.

If the channel is broken (because the authenticator process is stopped, or one of the m2_nodes that formed the channel path has stopped, or one of the TCP connections has been broken), the M2 network automatically generates a junkmail cancel message from the point of failure. Receiving this, the client sets the “established” signal to false.


Client / Component
Authenticator
Preconditions:
m2 connected
main session key generated
Authenticator service
available changes to true
Generate cookie
Encrypt cookie with authenticator public key
Encrypt session key with authenticator public key
Send to authenticator
req crypt_setup <e_session_key> <e_cookie>
Decrypt session key with authenticator private key
Log warning
No
Decrypt ok?
nack "Could not decrypt session key"
Yes
Decrypt cookie with authenticator private key
No
Decrypt ok?
nack "Could not decrypt cookie"
Yes
Associate session key with channel
pr_jm(session_id) ""
Generate session id and associate session key
with id
Decrypt cookie with session key
ack <e_cookie>
Encrypt cookie with session key
Cookie
matches?
No
Log warning
Yes
Set signal(established) to true
M2 connection lost,
authenticator shut down
jm_can(session_id) ""
Set signal(established) to false

Client Login

Once an encrypted channel is available (“established” state), the client can initiate a request to authenticate as a particular user, upgrading the connection to the “authenticated” state. This is the state where the authenticator has verified the client credentials and an RSA keypair has been generated for the user session, for use in federating the authentication to components.

Once the client software has user credentials (typically by prompting the user for them), it issues a request to the authenticator over the encrypted channel established earlier, containing the username and password to authenticate with.

On receiving this request, the authenticator checks if a session for this user already exists, and denies the login request if it does (a given user may only be logged in once).

The authenticator then delegates the user credential check to the plugin subsystem, which applies whatever logic the applicable plugin implements to verify the identity of the user. If this fails, the login request is denied.

If the user credential check passed, a new junkmail channel is established to the client, called the userinfo channel, and the preferences for this user are sent to the client.

The authenticator then checks if profiles are defined for this user. Profiles are named sets of attributes and permissions, which can vary for different roles (the user might have different permissions and configurations when acting in a different capacity for another team than in their normal role). If profiles are not defined, the authenticator uses the base set of attributes and permissions for this user.

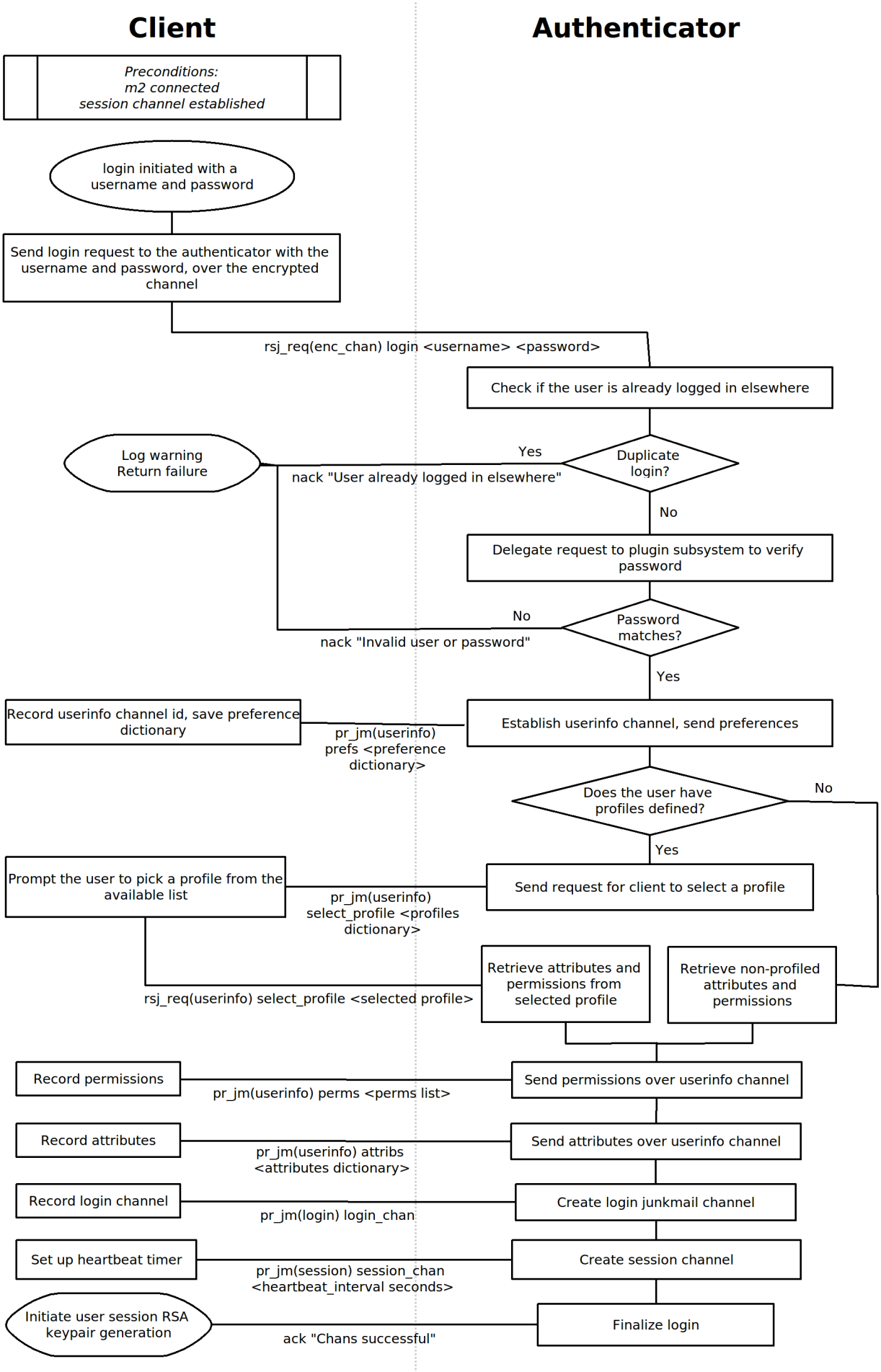
If profiles are defined, a message is sent back to the client requesting that the user select one of the available profiles. Typically the client software would prompt the user at this point for the profile to operate under. Once the user has selected the desired profile, a message to sent back to the authenticator with the selected profile, the authenticator retrieves the appropriate attributes and permissions and the login process proceeds.

A message is sent down the userinfo channel containing the permissions of the user, which is recorded by the client. Similarly a message is sent containing the user's attributes.

A new junkmail channel is established, called the login channel. This channel represents this identity's login to the system, and is used to track when all sessions for this identity have terminated (and therefore the identity is logged out). Since the authenticator enforces that a given user may only be logged in once the multi-session logic does not apply to users, but components, which authenticate in a similar fashion and also have a login channel set up are permitted multiple instances (to support patterns like load balanced or high availability services).

A new junkmail channel is established, called the session channel. This channel represents this particular instance of an identity's login to the system and is used to track this instances connection lifetime. In the event that an identity has more than one instance authenticated to the network as discussed above, then this is distinguished from the login channel in that a single login channel is subscribed to by all instances, whereas each instance has it's own session channel.

The request is then acknowledged by the authenticator. At this point the connection is authenticated from both sides – the client software has verified the identity of the authenticator software (through the use of the authenticator public key), and the authenticator has verified the identity of the user through the supplied credentials (typically a username and password).


Client
Authenticator
Preconditions:
m2 connected
session channel established
login initiated with a username and password
Send login request to the authenticator with the username and password, over the encrypted channel
rsj_req(enc_chan) login <username> <password>
Check if the user is already logged in elsewhere
Duplicate login?
Yes
nack "User already logged in elsewhere"
Log warning
Return failure
No
Delegate request to plugin subsystem to verify password
Password matches?
No
nack "Invalid user or password"
Yes
Establish userinfo channel, send preferences
pr_jm(userinfo) prefs <preference dictionary>
Record userinfo channel id, save preference dictionary
Does the user have profiles defined?
No
Yes
Send request for client to select a profile
pr_jm(userinfo) select_profile <profiles dictionary>
Prompt the user to pick a profile from the available list
rsj_req(userinfo) select_profile <selected profile>
Retrieve attributes and permissions from selected profile
Retrieve non-profiled attributes and permissions
Send permissions over userinfo channel
pr_jm(userinfo) perms <perms list>
Record permissions
Send attributes over userinfo channel
pr_jm(userinfo) attribs <attributes dictionary>
Record attributes
Create login junkmail channel
pr_jm(login) login_chan
Record login channel
Create session channel
pr_jm(session) session_chan <heartbeat_interval seconds>
Set up heartbeat timer
Finalize login
ack "Chans successful"
Initiate user session RSA keypair generation

User Session RSA Keypair Setup

After the above process completes successfully, the client then proceeds to generate an RSA keypair that is used later in the session to establish encrypted sessions and authenticate to components without requiring the user to separately supply their credentials to each component. This would be unacceptable for two reasons: it would be tedious for the user to do, as components are intended to provide single functions and the client will typically draw on the services of many components; and it would require the client to trust each component with their login information, or use different credentials for each one, so that a rogue component would not be able to impersonate the user to the rest of the network.

To address these issues, the client software generates a 1024 bit RSA keypair and sends the public key to the authenticator over the login channel. The authenticator records this public key for this user for use in later proxy authentications to components.

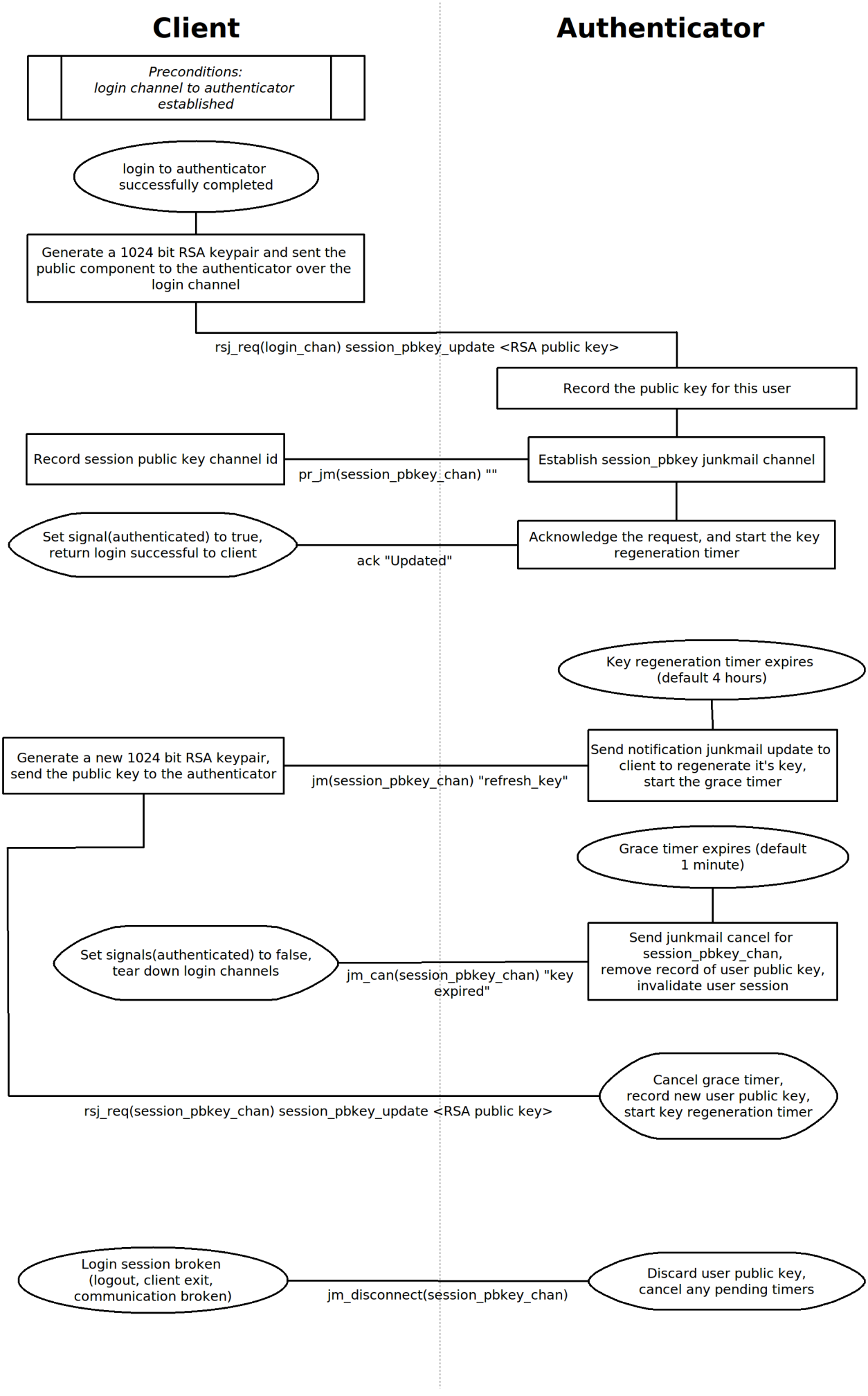
To address the issue of an adversary factoring the user's public key for long duration connections (no limit is imposed on the length of time a user may stay connected), a timer is started by the authenticator when it receives the public key, defaulting to 4 hours. When the timer expires, a junkmail update is sent to the client over the session_pbkey channel requesting that the client generate a new keypair. A grace timeout is started, defaulting to 1 minute.

The client then generates a new keypair, and sends the public key to the authenticator over the session_pbkey channel. On receiving this, the authenticator updates the public key stored for this user, cancels the grace timeout and starts a new key regeneration timeout.

If the client does not respond within the grace time, the authenticator discards the stored key for the user and cancels the session.

When the channel is broken from the client side, due to logout, client software exit, network connection failure or any other reason, the authenticator discards the stored key and cancels the session.

It is believed that, barring mathematical breakthroughs in factoring large numbers, a 1024 bit RSA key will be sufficiently strong for the permitted validity period for the forseeable future. Should it become computationally feasible to break the key inside the default 4 hour validity period, the time can be reduced as needed (without change to deployed client software, since the time is controlled by the authenticator).


Client
Authenticator
Preconditions:
login channel to authenticator
established
login to authenticator
successfully completed
Generate a 1024 bit RSA keypair and sent the
public component to the authenticator over the
login channel
rsj_req(login_chan) session_pbkey_update <RSA public key>
Record the public key for this user
Record session public key channel id
pr_jm(session_pbkey_chan) ""
Establish session_pbkey junkmail channel
Set signal(authenticated) to true,
return login successful to client
ack "Updated"
Acknowledge the request, and start the key
regeneration timer
Key regeneration timer expires
(default 4 hours)
Generate a new 1024 bit RSA keypair,
send the public key to the authenticator
jm(session_pbkey_chan) "refresh_key"
Send notification junkmail update to
client to regenerate it's key,
start the grace timer
Grace timer expires (default
1 minute)
Set signals(authenticated) to false,
tear down login channels
jm_can(session_pbkey_chan) "key
expired"
Send junkmail cancel for
session_pbkey_chan,
remove record of user public key,
invalidate user session
Cancel grace timer,
record new user public key,
start key regeneration timer
rsj_req(session_pbkey_chan) session_pbkey_update <RSA public key>
Login session broken
(logout, client exit,
communication broken)
jm_disconnect(session_pbkey_chan)
Discard user public key,
cancel any pending timers

Component Login

A component can authenticate to the network in a similar fashion to a client, in order to consume services from other components, although this is not required in order to provide services.

In contrast to a user login, the component authenticates itself using the keypair associated with its service tag.

The component first requests a cookie from the authenticator, which it generates from its CSPRNG along with an ID, which are stored temporarily. These are sent back as the response to the request to the component.

The component encrypts the cookie using the private key of the service it wants to authenticate as, and sends this signed (encrypted) cookie together with the cookie ID and the service tag it is authenticating as to the authenticator.

The authenticator keeps a registry of the public key associated with each service tag. It retrieves the public key corresponding to the service tag in the login request, decrypts the signed cookie using the public key, and compares it with the cookie saved against the supplied cookie ID.

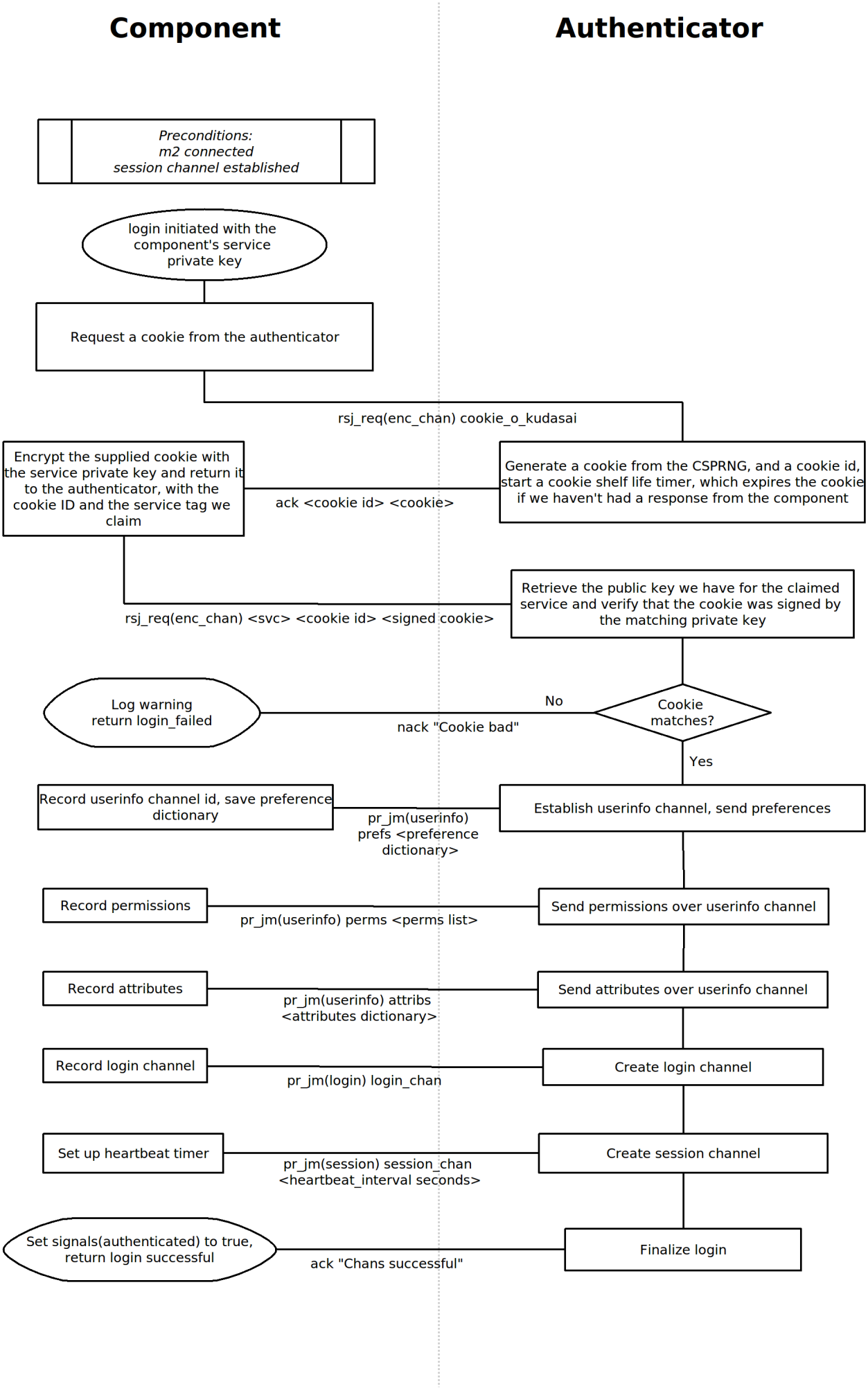
If the cookie does not match, the request is denied and the login attempt fails.

If the cookie matches the preferences, permissions and attribute userinfo channels are established as for the client login, with the exception that profiled attributes and permissions are not supported for component logins.

A login channel is established (or joined if there already is a login session for this service from another instance).

A session channel is established, unique for this login instance.

The login request is acknowledged, and the client enters the “authenticated” state, and returns login successful from the login function.


Component
Authenticator
Preconditions:
m2 connected
session channel established
login initiated with the component's service private key
Request a cookie from the authenticator
rsj_req(enc_chan) cookie_o_kudasai
Generate a cookie from the CSPRNG, and a cookie id, start a cookie shelf life timer, which expires the cookie if we haven't had a response from the component
ack <cookie id> <cookie>
Encrypt the supplied cookie with the service private key and return it to the authenticator, with the cookie ID and the service tag we claim
rsj_req(enc_chan) <svc> <cookie id> <signed cookie>
Retrieve the public key we have for the claimed service and verify that the cookie was signed by the matching private key
Cookie matches?
No
nack "Cookie bad"
Log warning
return login_failed
Yes
Establish userinfo channel, send preferences
pr_jm(userinfo) prefs <preference dictionary>
Record userinfo channel id, save preference dictionary
Send permissions over userinfo channel
pr_jm(userinfo) perms <perms list>
Record permissions
Send attributes over userinfo channel
pr_jm(userinfo) attribs <attributes dictionary>
Record attributes
Create login channel
pr_jm(login) login_chan
Record login channel
Create session channel
pr_jm(session) session_chan <heartbeat_interval seconds>
Set up heartbeat timer
Finalize login
ack "Chans successful"
Set signals(authenticated) to true, return login successful

Client to Component Authentication

Once the applicable processes described above have been completed, the client can connect to services hosted by components, with encryption and authentication automatically managed behind the scenes at connect time. The processes described below detail this mechanism.

From the client side the connection is managed by a class called Connector, which acts as a proxy within the client for a service provided by a component elsewhere on the network. The following diagram sets the processes involved in their state network context.

In typical operation, the state network proceeds as follows:

The “available” signal, managed by the M2 api class, an ancestor of the Authenticator class, is arranged to represent the availability of the service with which a connection is desired. It will usually already be true, since the service is typically continually provided, whereas the connector object is newly constructed.

The “authenticated” signal, managed by the Authenticator class, will typically start out false, since the connector object is usually created at client start time, before the user has logged in. Once the user login is successful, the signal transitions to true. This initiates the process to request the service's registered public key from the authenticator (described in flowchart <TODO ref>). If this process resolves in the public key being retrieved, the “got_svc_pubkey” signal (managed by the Connector class) is set to true.

The Connector class creates an AND gate called “connect_ready” to group the preconditions to establishing a connection to the service together. The inputs are the “available”, “authenticated” and “got_svc_pubkey” signals described above. When all three of these are true, the gate's output transitions to true, reflecting the fact that all the requirements for establishing a connection to the service are in place.

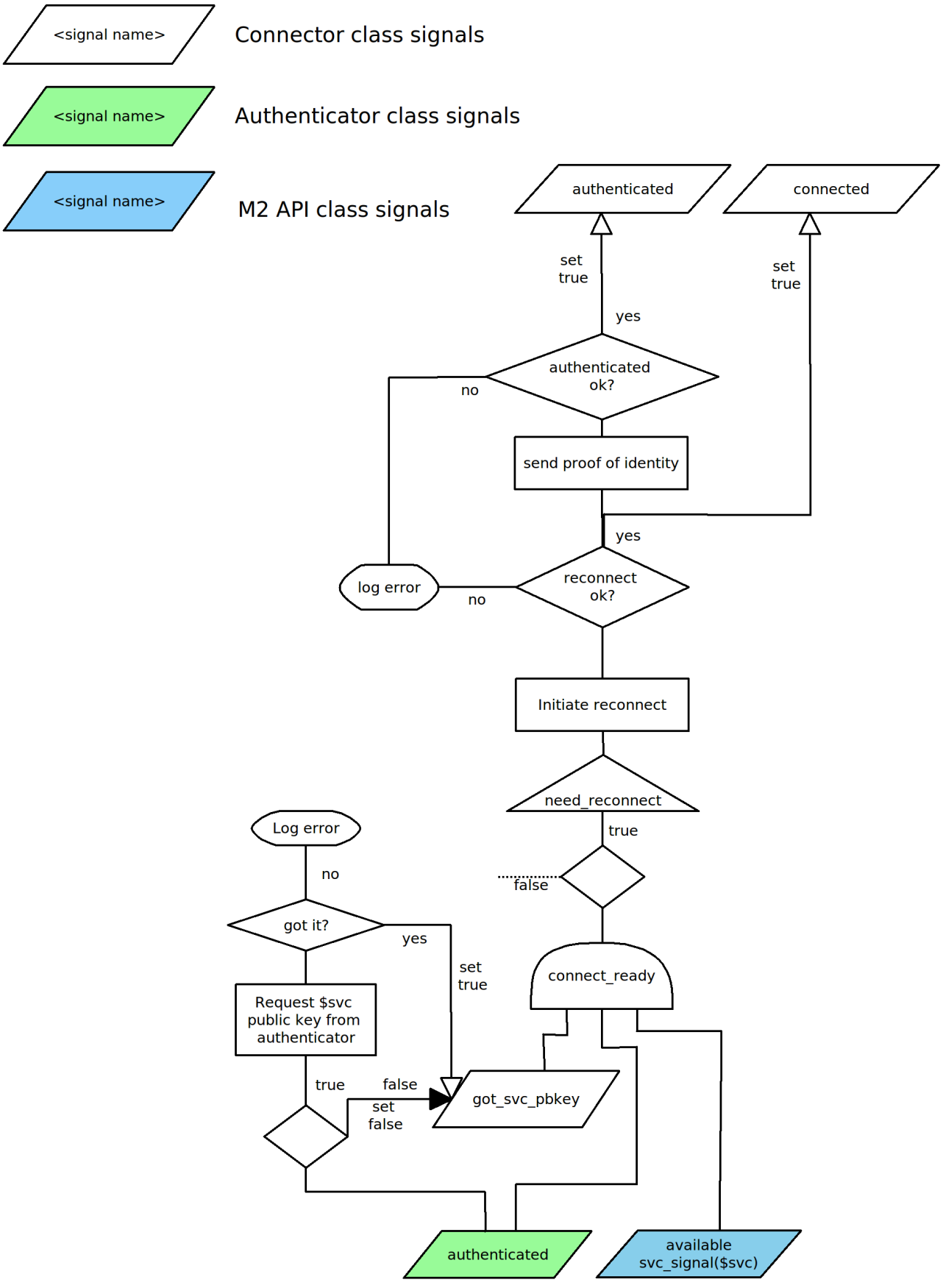
The output of the “connect_ready” gate is fed through a domino object, which initiates the reconnection process (described in <TODO ref>) a short time in the future (typically microseconds).

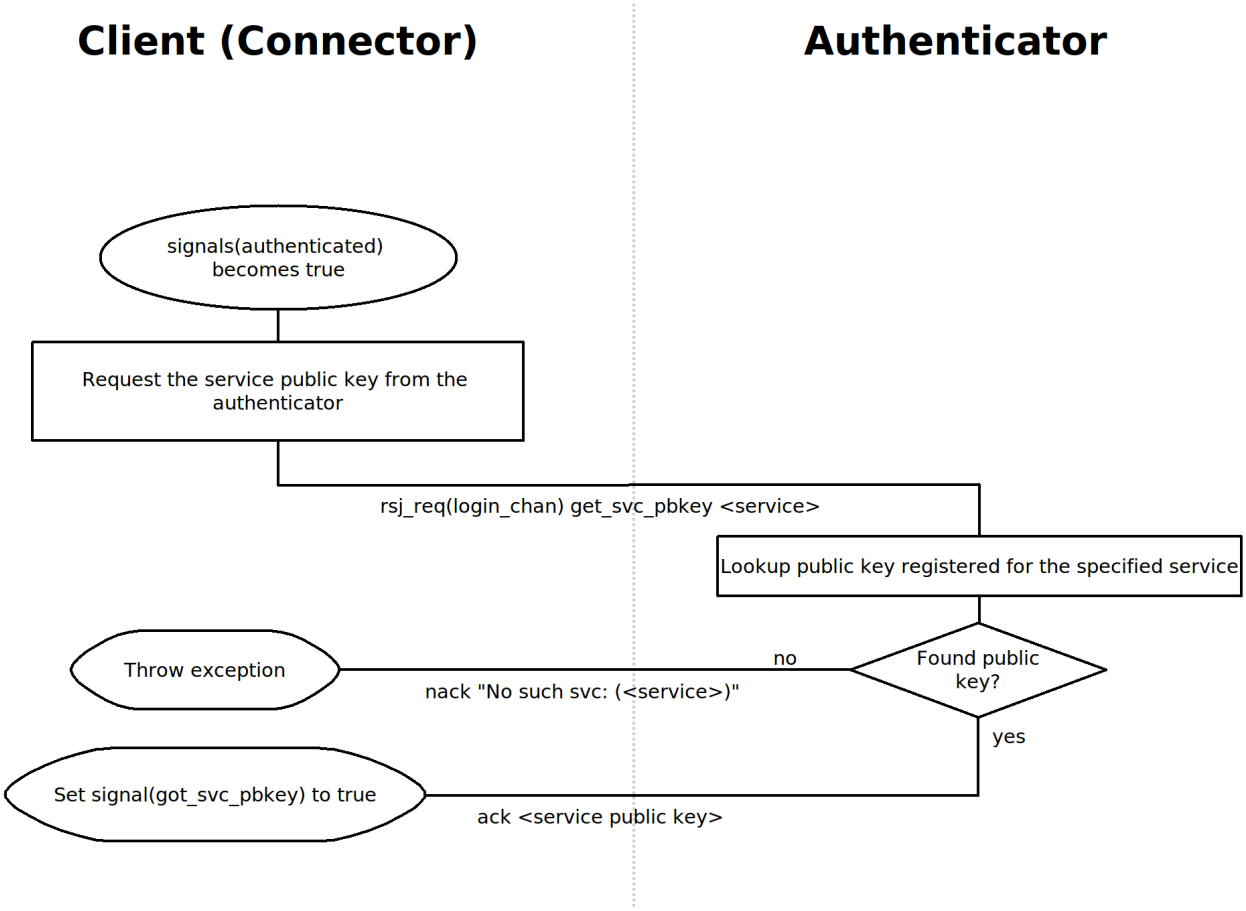
The reconnect process, if successful, establishes an encrypted channel with the component hosting the service, using the service's public key to securely send it a session key generated for the connection. The Connector class sets the “connected” signal to true, representing the fact that an encrypted channel has been established.

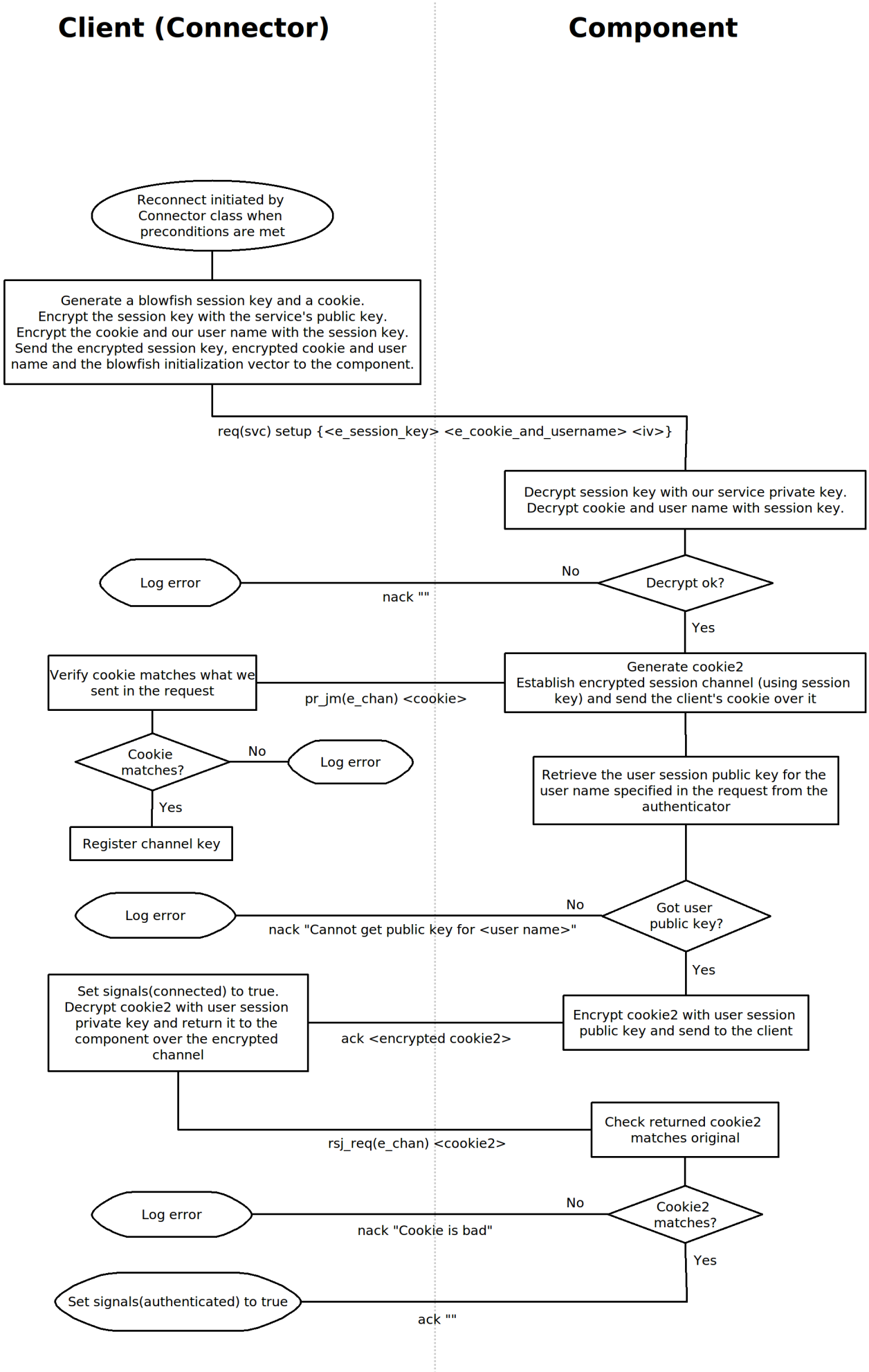
The component then sends the client a cookie over the encrypted channel, which the client signs using the user session RSA private key and sends back as proof of identity to the component (as described in <TODO ref>). If this proof is accepted, verified by the component by retrieving the active user RSA public key from the authenticator, the Connector class sets it's “authenticated” signal to true.

Application logic in the client that wishes to interact with the service typically hooks into the state of the connector's “authenticated” signal to know when to initiate it's communication.

Connector Class State System




Client (Connector)
Authenticator
signals(authenticated) becomes true
Request the service public key from the authenticator
rsj_req(login_chan) get_svc_pbkey <service>
Lookup public key registered for the specified service
Found public key?
no
Throw exception
nack "No such svc: (<service>)"
yes
Set signal(got_svc_pbkey) to true
ack <service public key>


Client (Connector)
Component
Reconnect initiated by Connector class when preconditions are met
Generate a blowfish session key and a cookie. Encrypt the session key with the service's public key. Encrypt the cookie and our user name with the session key. Send the encrypted session key, encrypted cookie and user name and the blowfish initialization vector to the component.
req(svc) setup {<e_session_key> <e_cookie_and_username> <iv>}
Decrypt session key with our service private key. Decrypt cookie and user name with session key.
Decrypt ok?
No
Log error
nack ""
Yes
Generate cookie2 Establish encrypted session channel (using session key) and send the client's cookie over it
pr_jm(e_chan) <cookie>
Verify cookie matches what we sent in the request
Cookie matches?
No
Log error
Yes
Register channel key
Retrieve the user session public key for the user name specified in the request from the authenticator
Got user public key?
No
Log error
nack "Cannot get public key for <user name>"
Yes
Encrypt cookie2 with user session public key and send to the client
ack <encrypted cookie2>
Set signals(connected) to true. Decrypt cookie2 with user session private key and return it to the component over the encrypted channel
rsj_req(e_chan) <cookie2>
Check returned cookie2 matches original
Cookie2 matches?
No
Log error
nack "Cookie is bad"
Yes
Set signals(authenticated) to true
ack ""