Seif Protocol

Excited to share our progress on project seif. We will be delving deep into Seif protocol which will enable secure JSON over TCP.

Why Seif?

— The main culprit is really that todays web hardly conforms to its intended design.
— http request response protocol designed to retrieve documents — has negotiation
— SSL — link encryption. SSL is very complicated. Reliance on certificate authorities, who to trust? Should we trust anyone at all? Do we need a certificate authority?
— HTML mark up language for documents — javascript — DOM — CSS.
— Adding work arounds increases complexity. Reasoning then about security/correctness of a complex system.
— Web works — best open application delivery system around .
— Seif is an open initiative towards enabling a safe and effective relationship on the web. It is intended to coexist with the web through a helper app on the browser.
— Idea is to make it easy to get stuff right and difficult to not so.

The Journey

Breaking down seif project into milestones.
Step 1: Seifnode — Cryptographic services for node servers.
Step 2: Seif Protocol — Secure JSON over TCP
Step 3: Seif Resource Management — hash based resource retrieval
Step 4: Seif Apps — Qt QML based widgets as applications, no HTML
Step 5: Helper app

seifnode

Download: https://github.com/paypal/seifnode

Seifnode is a step at providing reliable crypto services to node servers. Its main contribution is its random number generator seeded with entropy sources that are uncorrelated and highly random.

- Random
- ECC 521
- AES 256
- SHA3-256

seifnode

Download: https://github.com/paypal/seifnode

Explain crypto services

…seif protocol

The main motivation for the seif protocol, more so project seif is to keep things simple such that reasoning about reliability, security and correctness is possible.

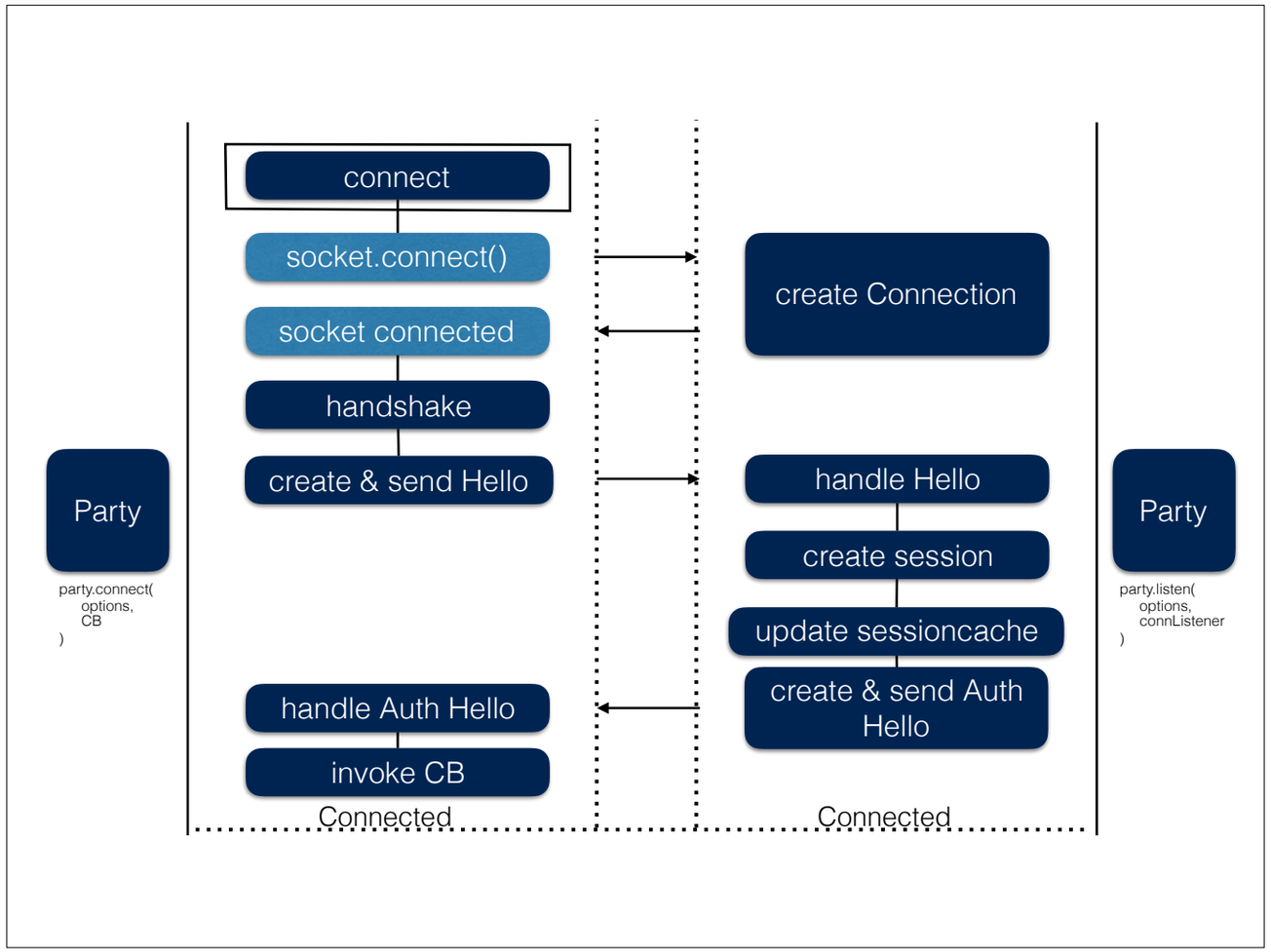
The protocol has no negotiation, dependence on certificate authorities. Essentially simplifying connections through fewer handshakes and ousting third party entities.

It is a message delivery protocol, doing away with the traditional request-response paradigm.

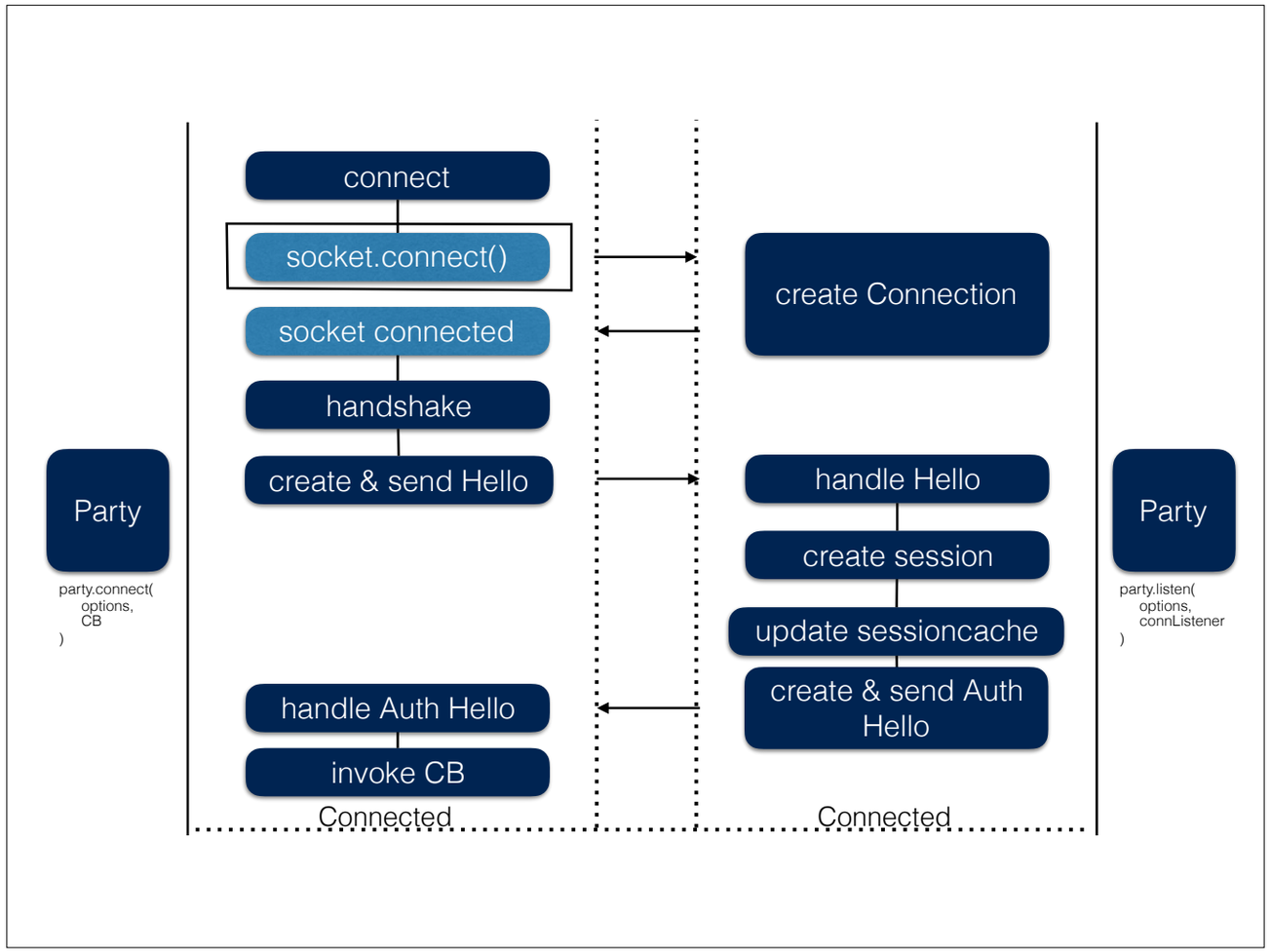
It is built on seifnode enabling reliable link encryption and random services. First contact relies on Public key cryptography for key exchange.

Disk encryption for keys, state

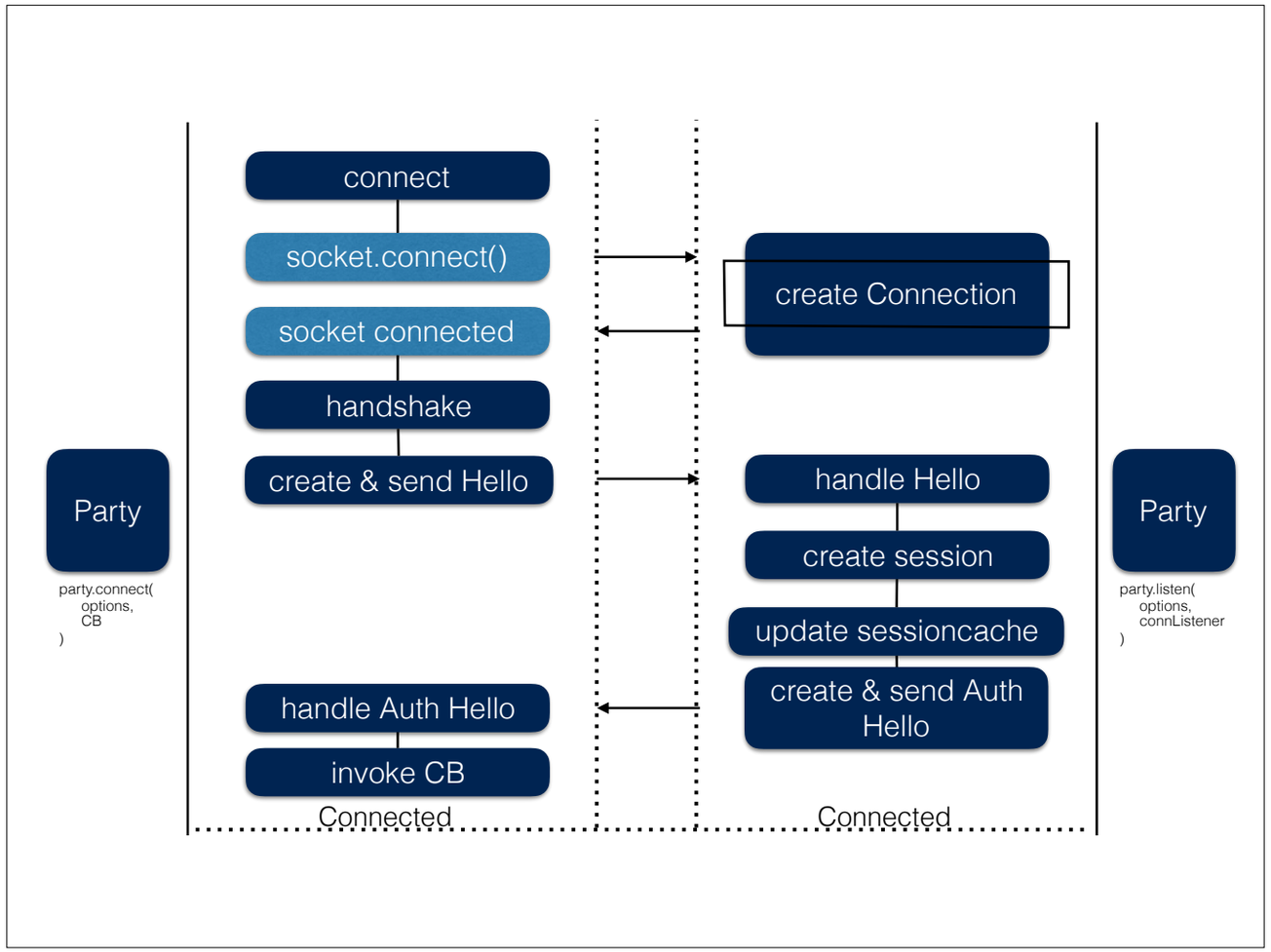
-broad description given
-getting into the details next
-understand the broad ideas behind the protocol better
-party - what is a party? representation of the application entity at seif layer, just like a TCP socket
-Party represents an identity which is a combination of the public/private key pair and the RNG state
-First step is to initialize a party. This can be done using a simple username/password. The protocol requires the hash of the password to be used to decrypt the public/private keys and the private key to be used to decrypt the RNG and other state stored on the disk etc.



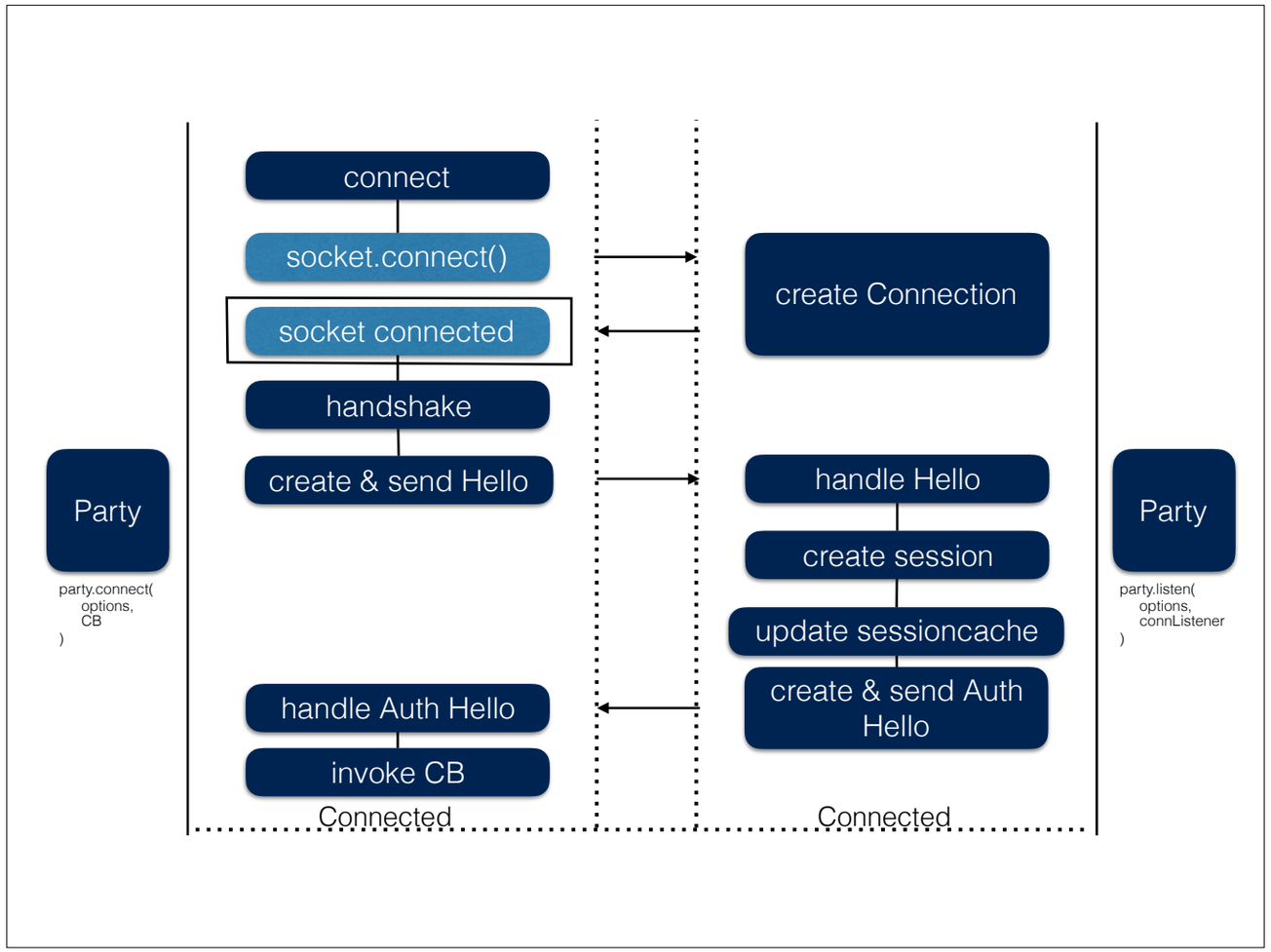
- This is a high level flow diagram of the steps involved setting up a connection between 2 parties.
- describe the protocol flow for the simple case-where a party is connecting to another for the first time..so therefore no session exists
- rhs - listen, lhs - connect
- params that listen can take and connect can take



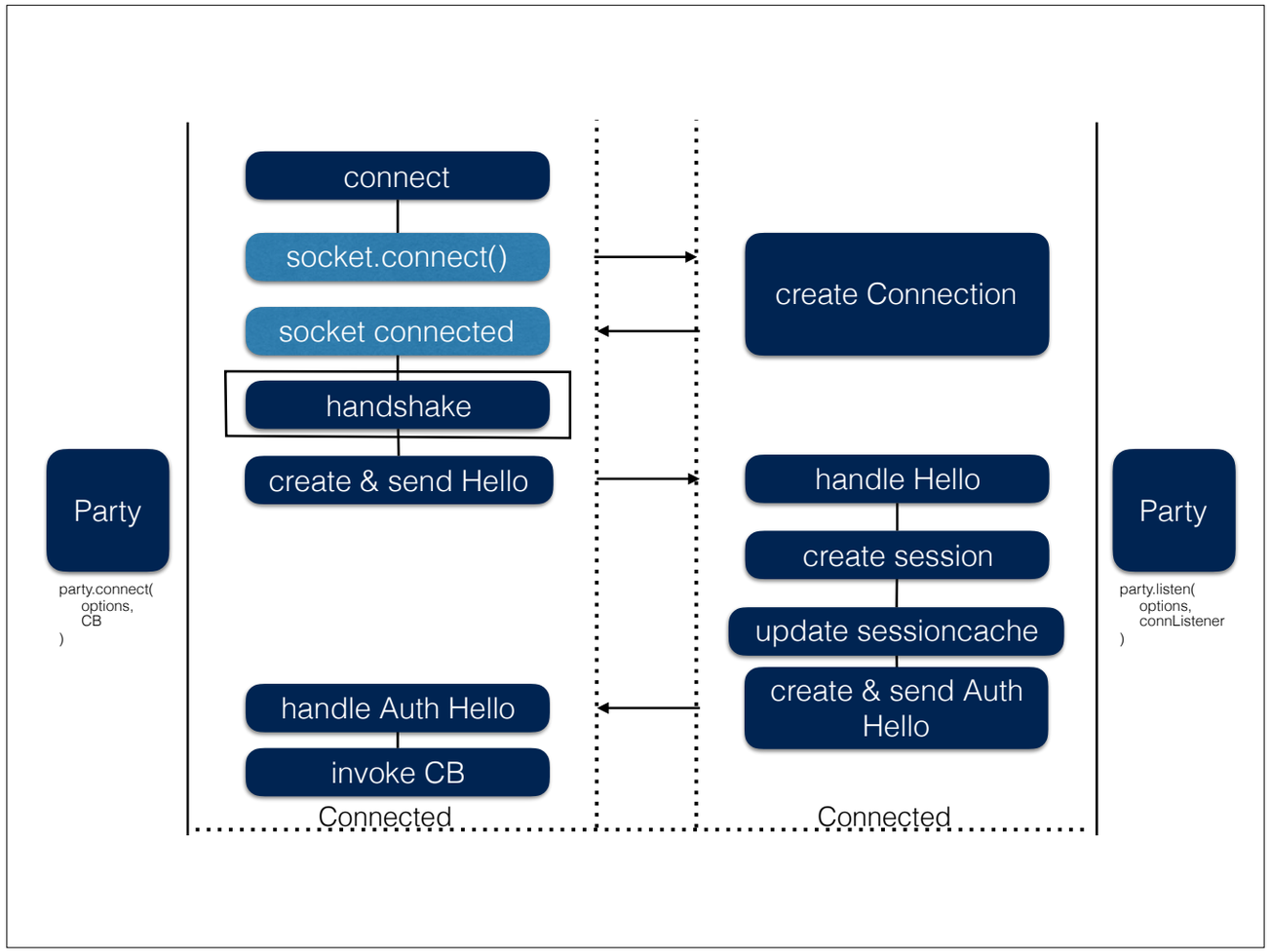
- Initiate a TCP connection



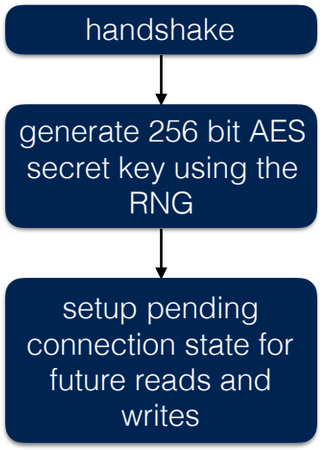
- Receive tcp connection and the associated socket



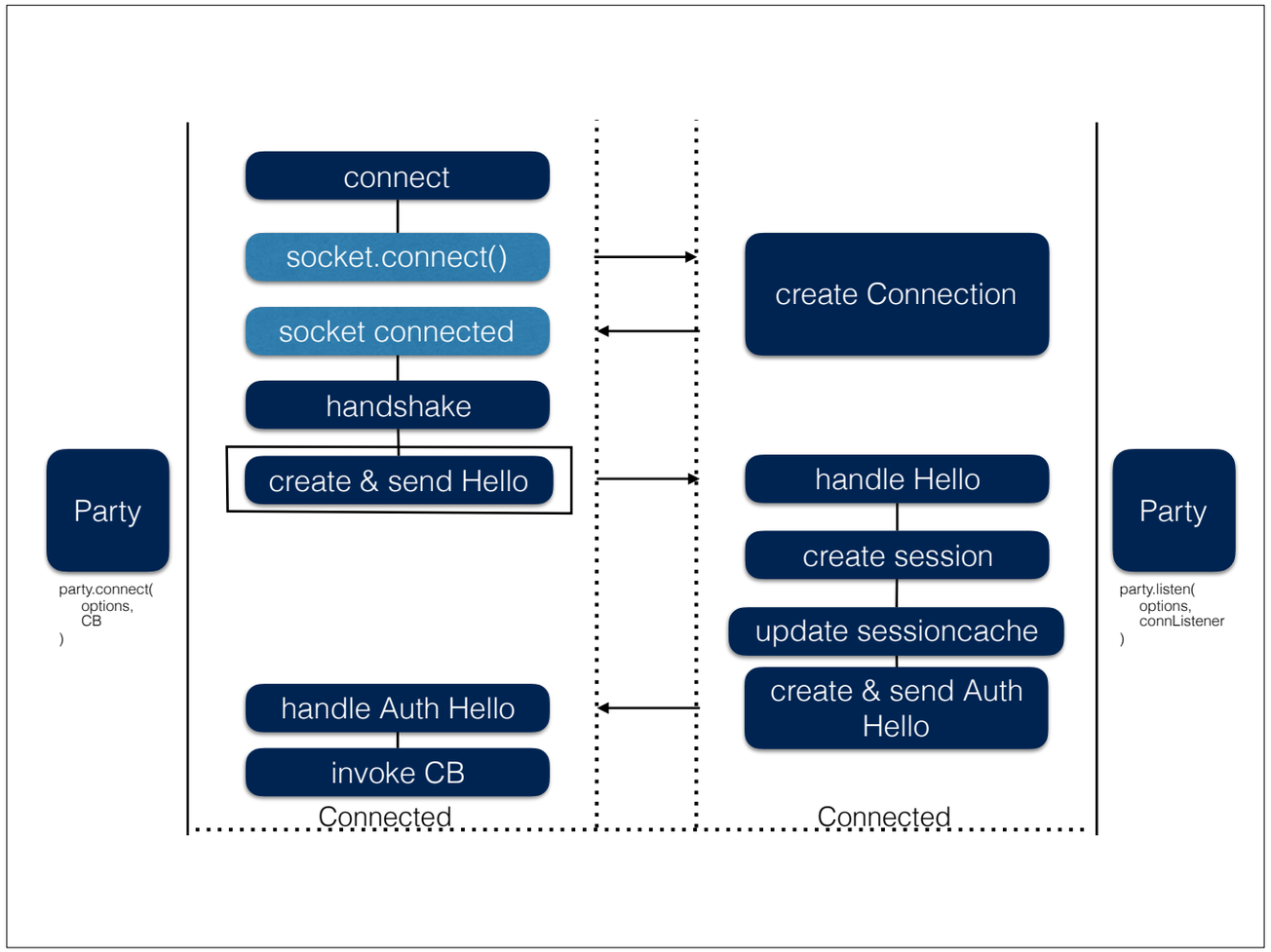
- Initiator receives “connected” event on its socket



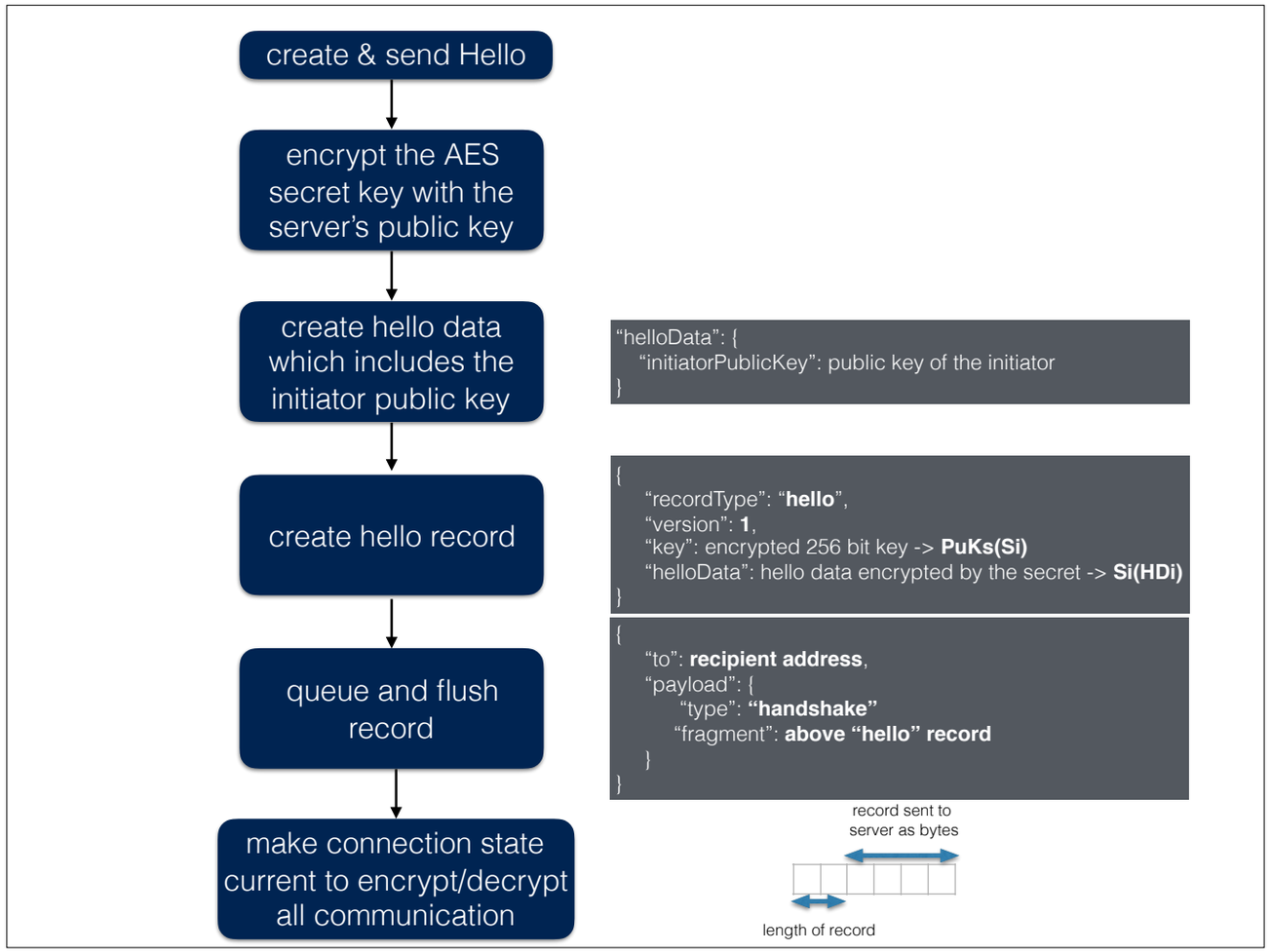
- Now that the TCP connection has been established, initiate the process to establish the seif connection
- Handshake process begins



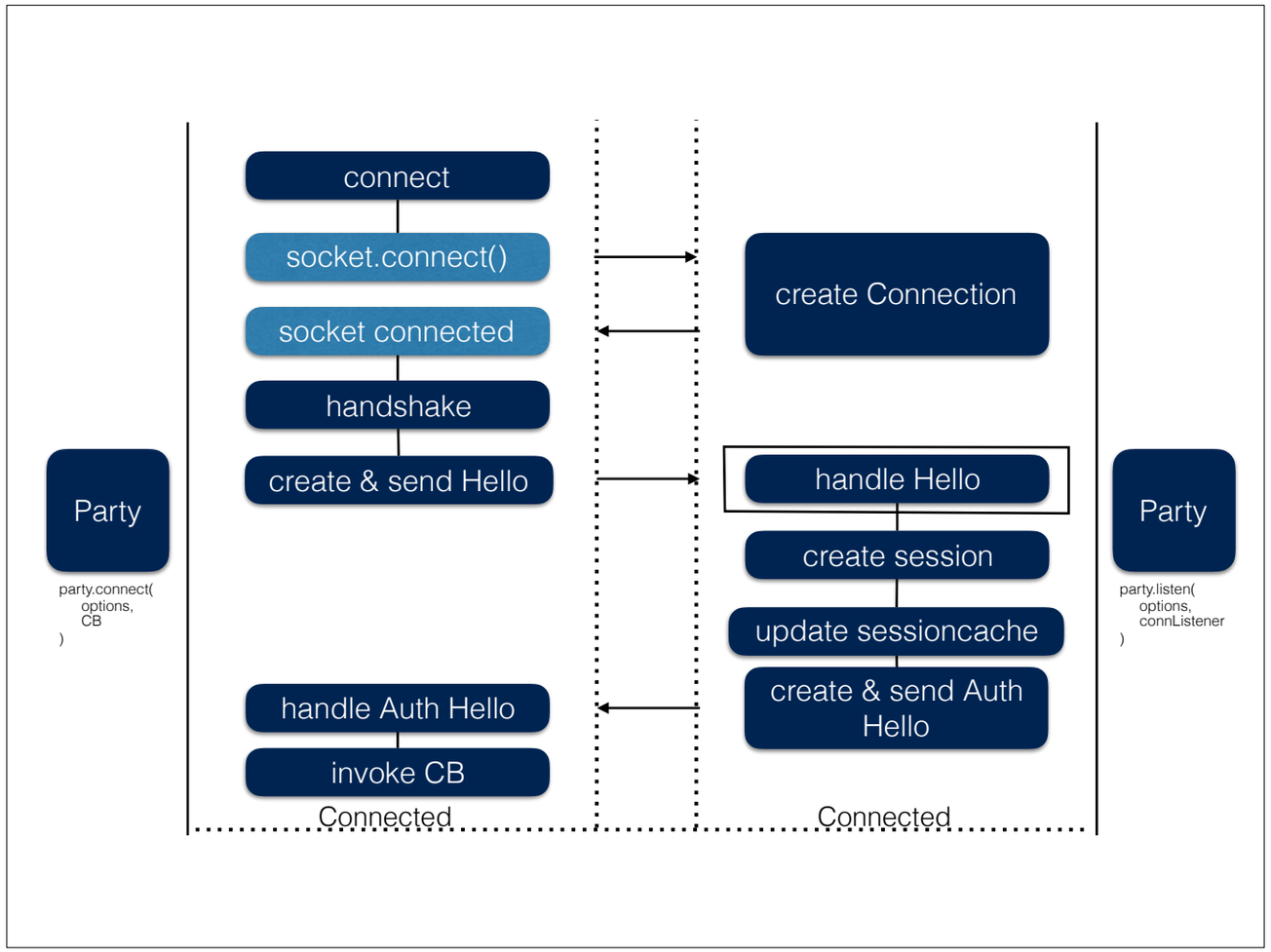
- Generate a 256 bit AES secret key using the seif RNG
- Setup a pending connection state - (read/write modes). This is pending because the first seif connection message is going to be sent in the clear and this is going to be used for future reads and writes.



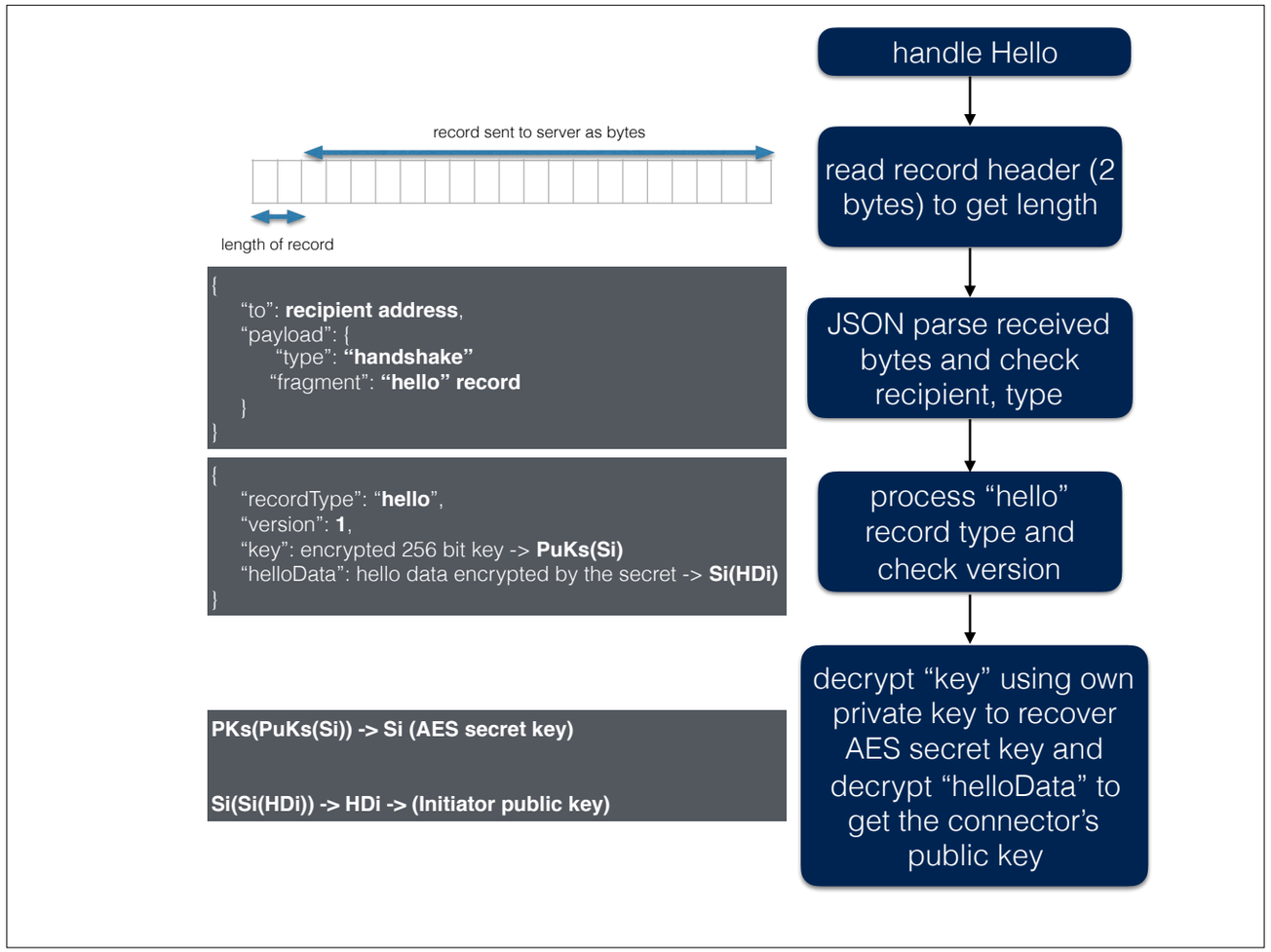
- Create first hello record



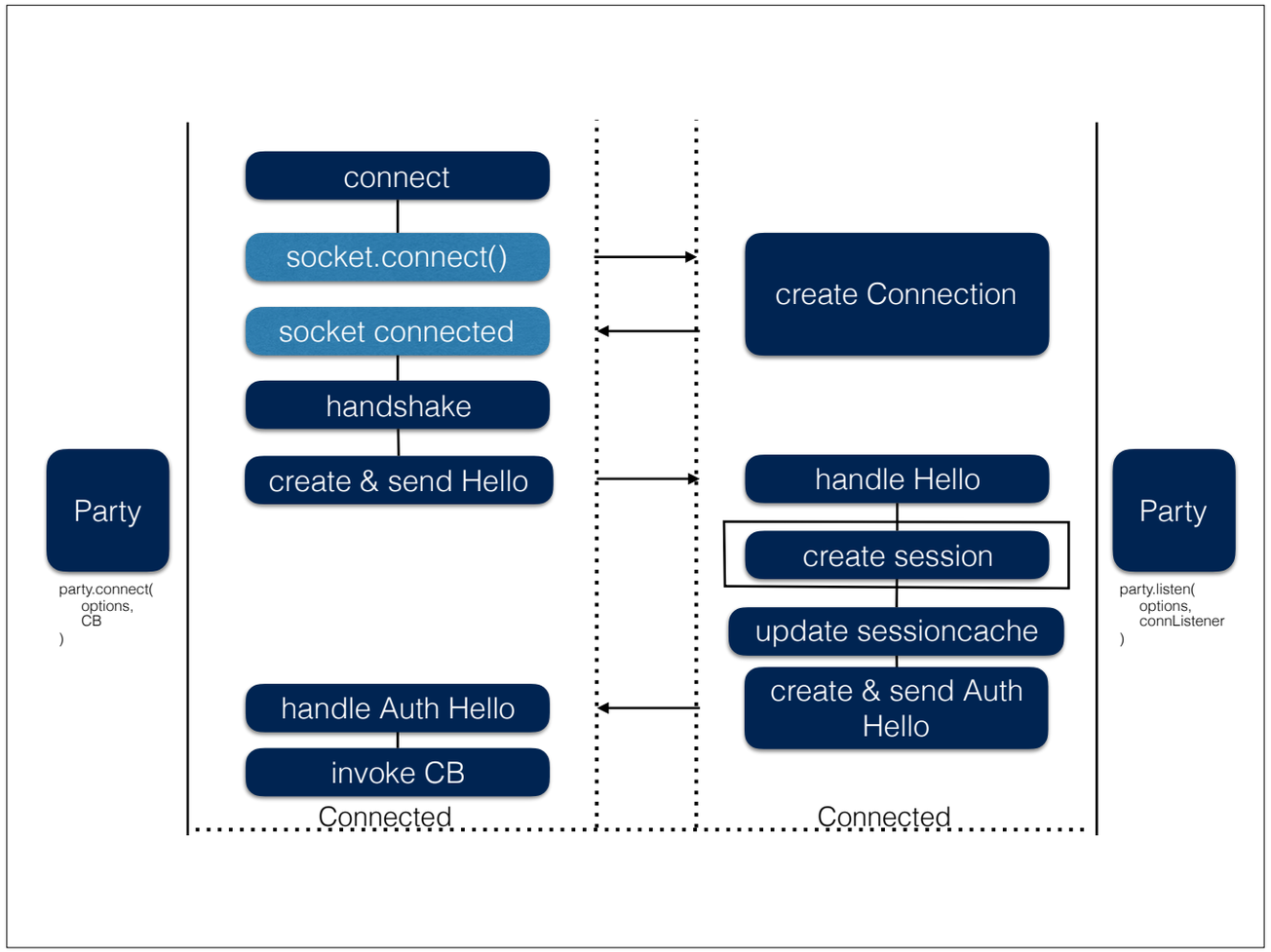
- Encrypt the AES key using the server's public key
- Initialize the helloData using the initiator's public key. This object is used for other more complex cases.
- Encrypt the helloData record using the AES secret key
- Wrap the hello record into a seif record with recipient and type information indicating a handshake
- Flush the record on the tcp connection with a length indicator of 2 bytes preceding it
- Make connection state current. So now the party expects the future communication to be encrypted

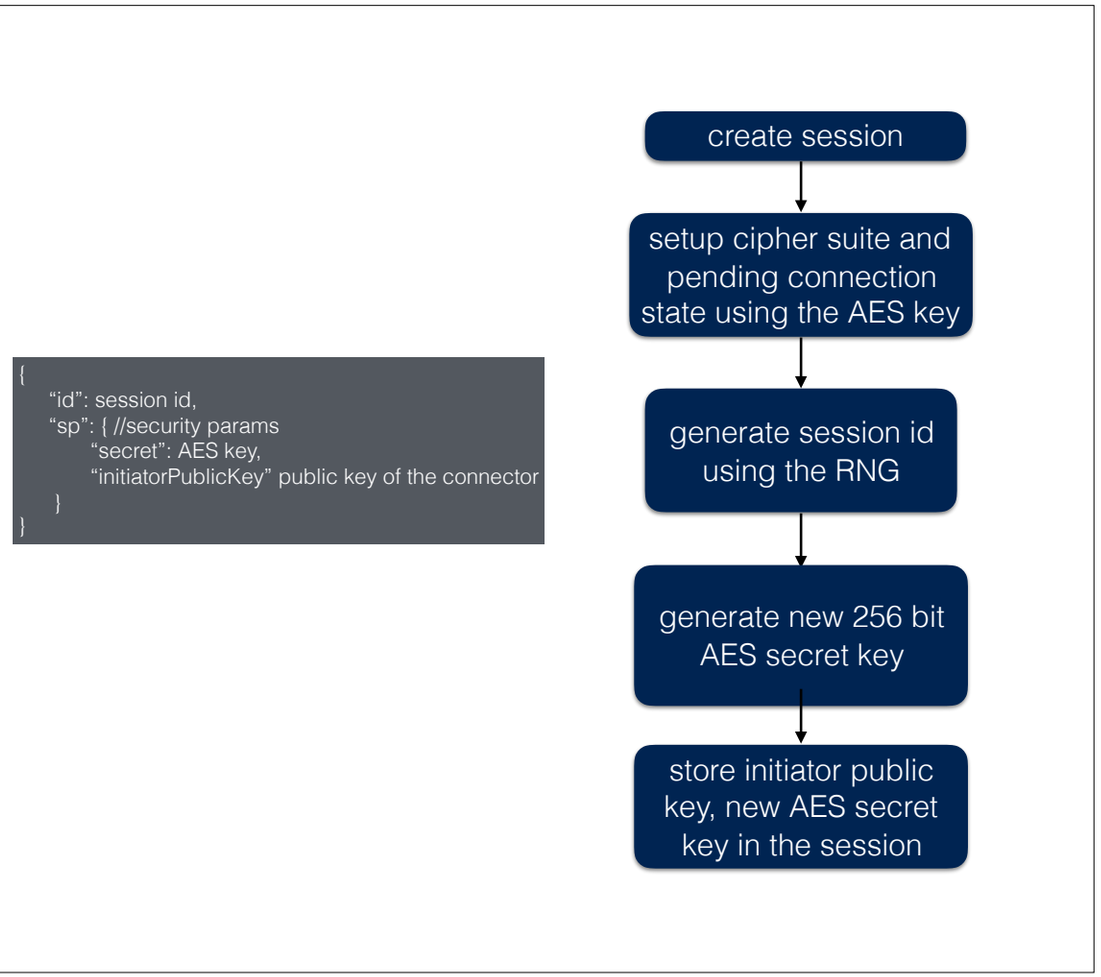


- TCP data received by listening party
- Data ready to be processed

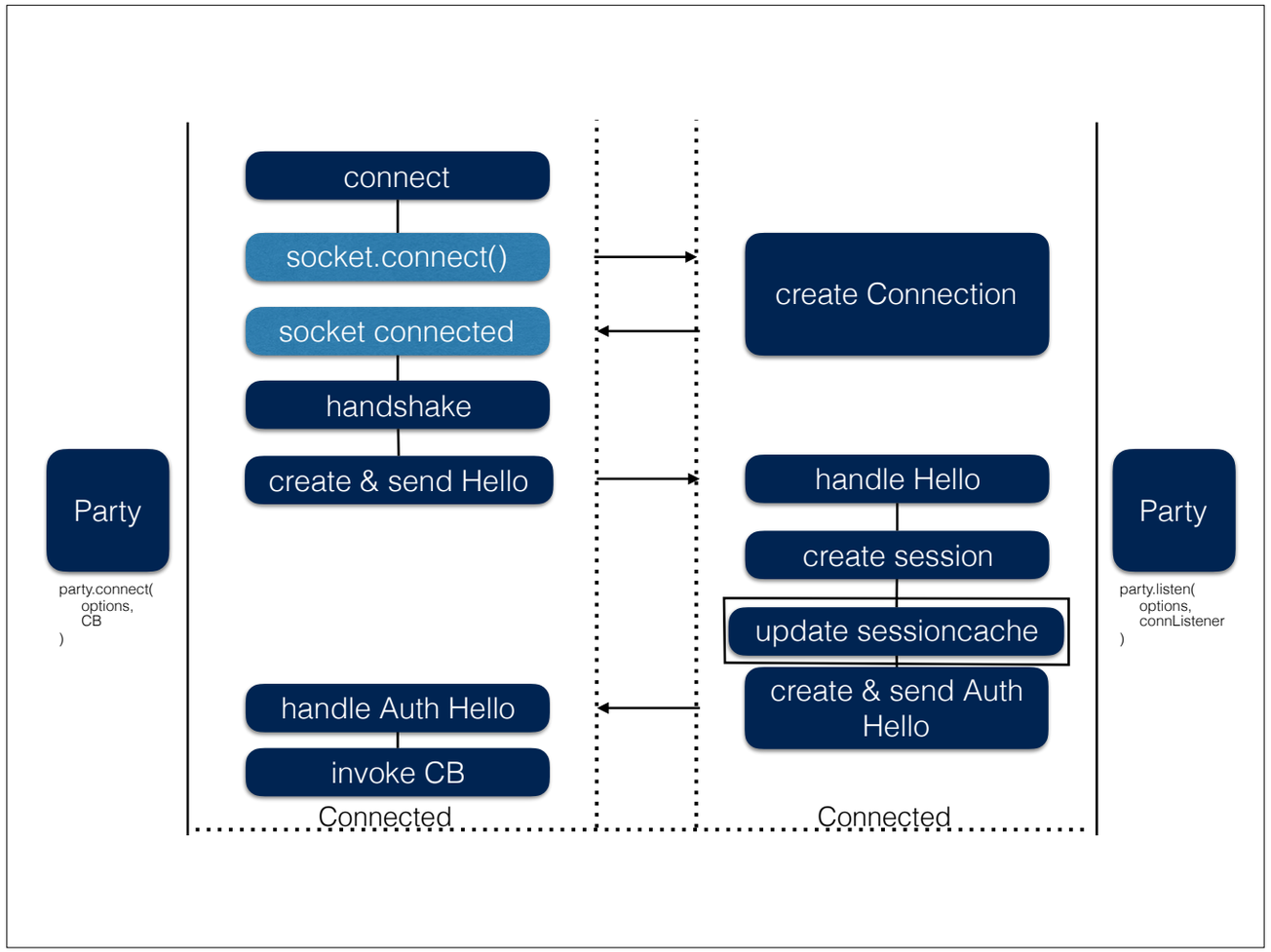


- Read the length from the first 2 bytes and now the party knows how many bytes to process
- JSON parse the record and perform initial validation
- Unwrap the seif record to get the underlying “hello” record
- Decrypt the “key” and the “helloData” to get the AES secret key and the initiator’s public key

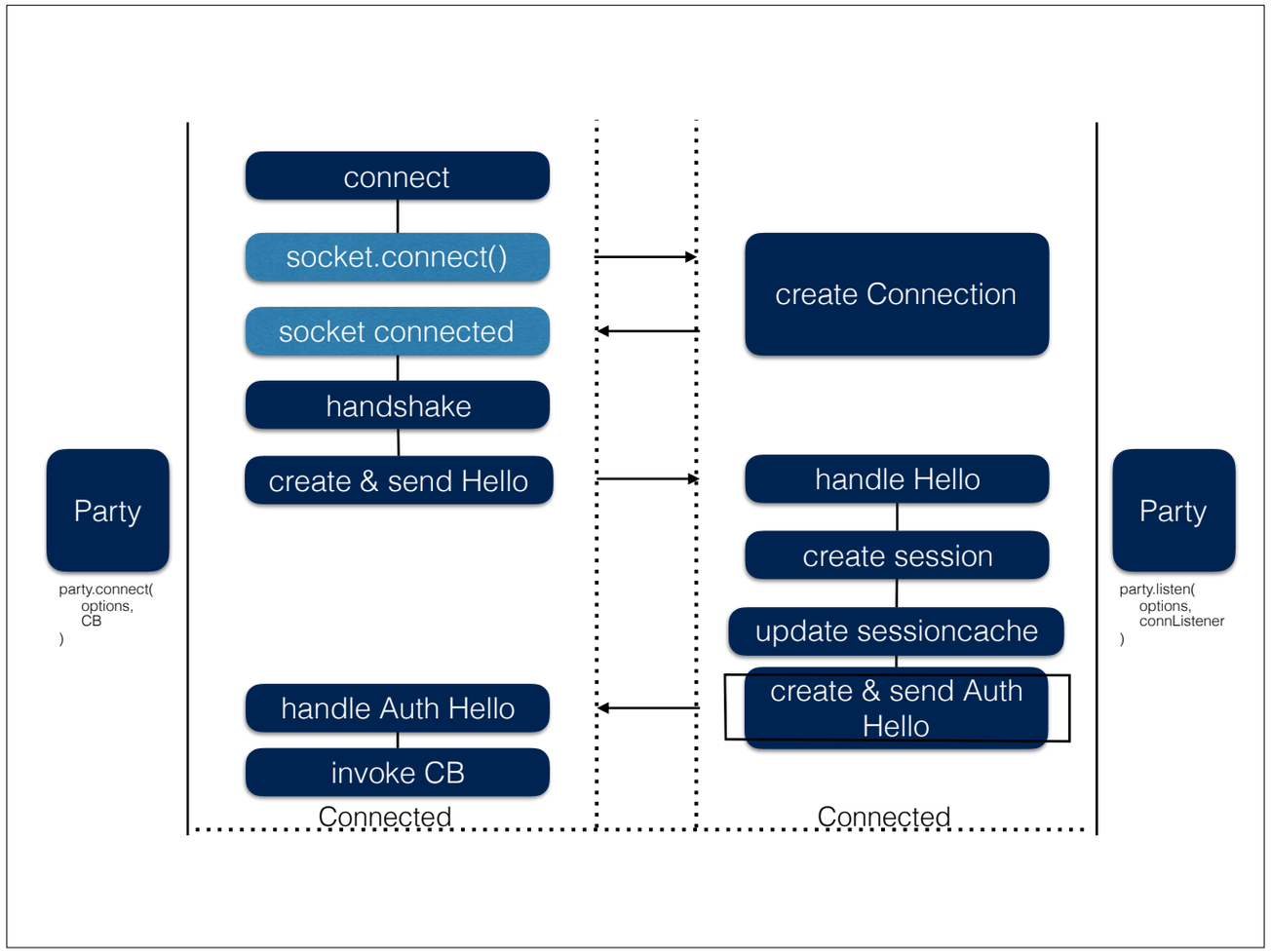

connect
socket.connect()
socket connected
handshake
create & send Hello
handle Auth Hello
invoke CB
Party
party.connect(
options,
CB
)
create Connection
handle Hello
create session
update sessioncache
create & send Auth Hello
Party
party.listen(
options,
connListener
)
Connected
Connected



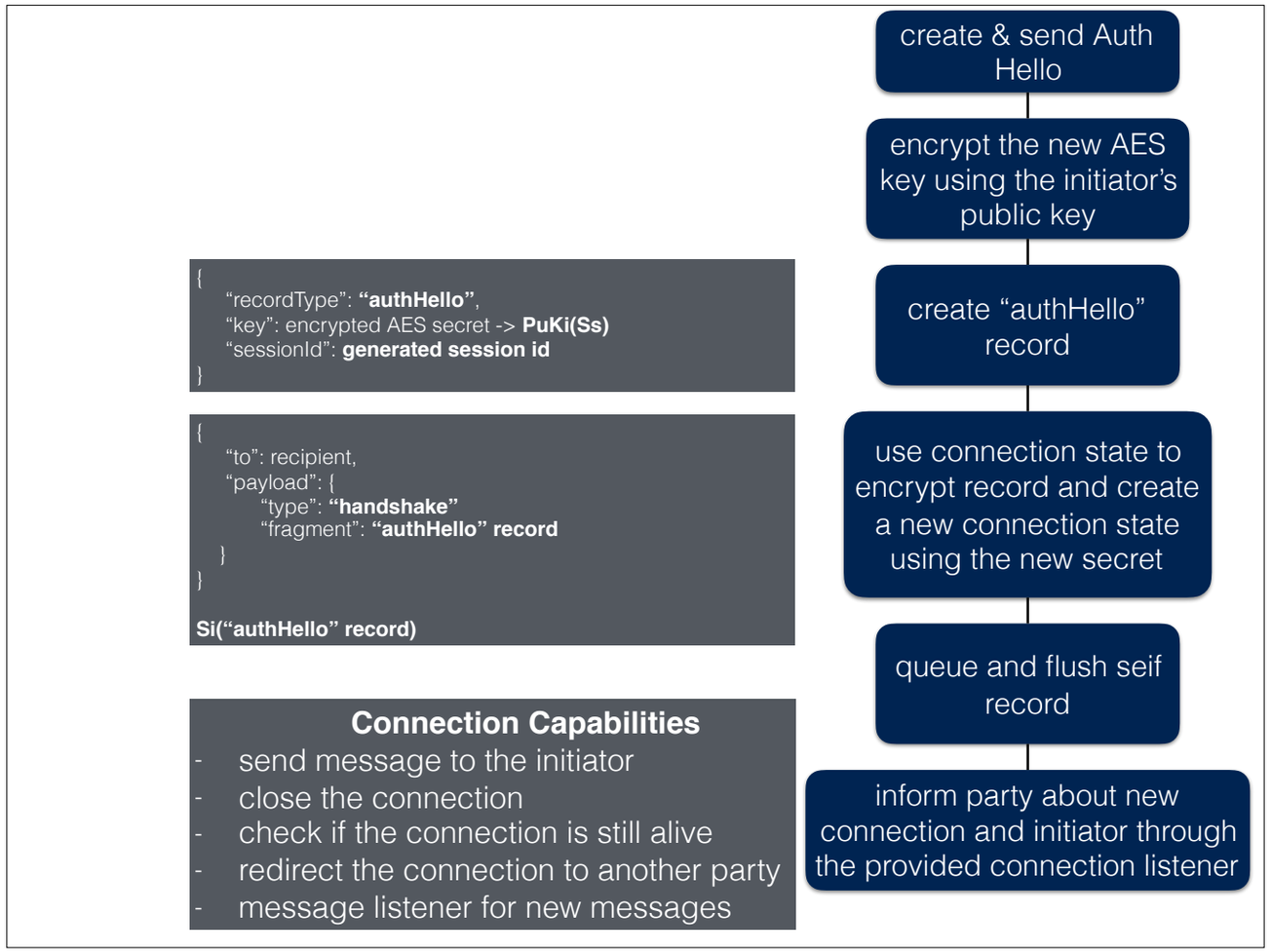
- Before creating session, create the connection state using the received AES secret key
- Generate a new session identifier and a new 256 bit AES secret key using the RNG
- Store the id and the security params including the key and the public key in the session



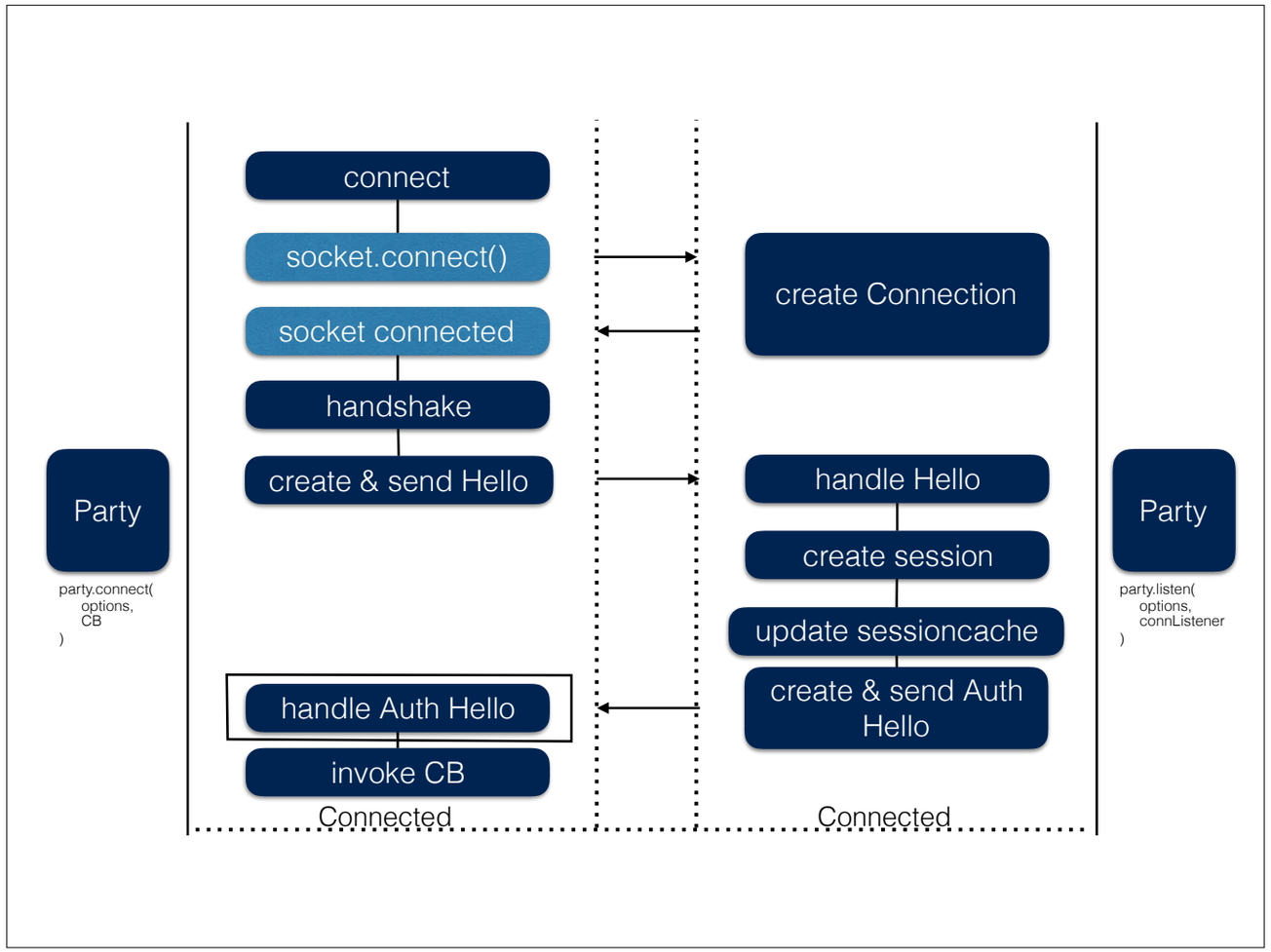
- Update the session cache with the id to session object mapping
- The protocol requires the values in the session cache to be AES encrypted using the hash of the private key

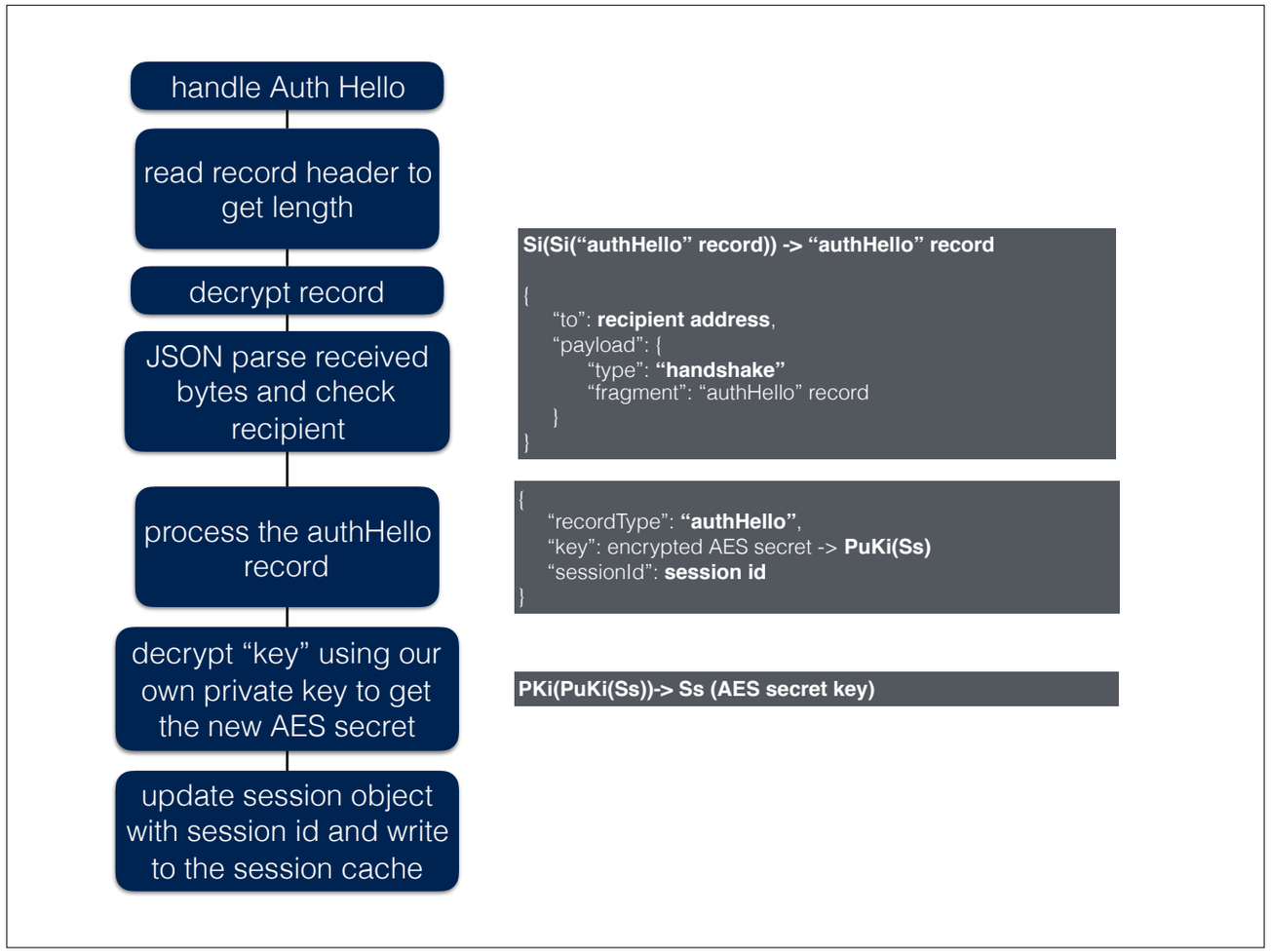


- Create auth hello record
- purpose of this record is to authenticate the initiator

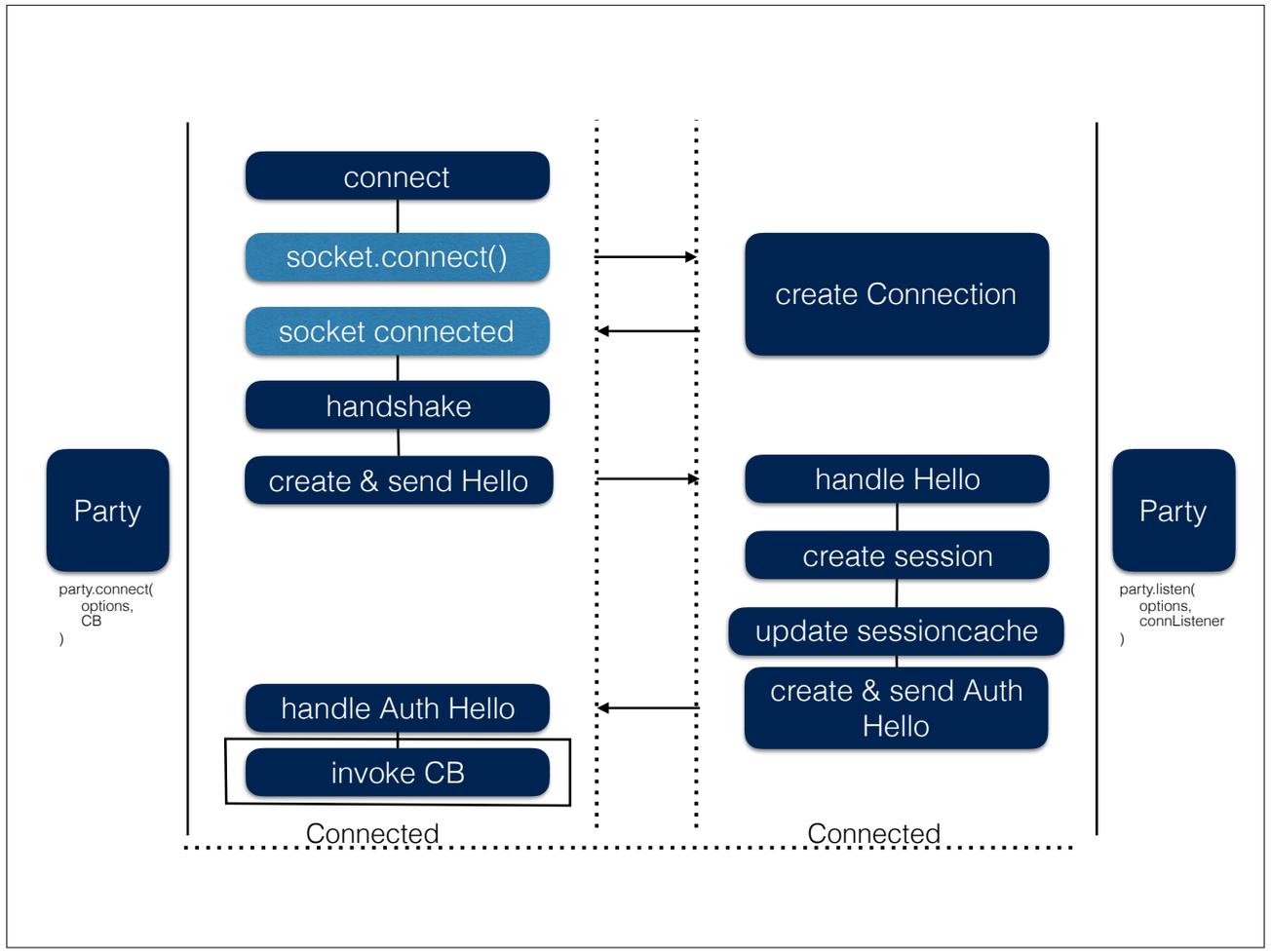


- Encrypt the newly generated AES key using the initiator's public key
- Create the record using the above encrypted key and the generated session id
- Wrap this into the seif record and flush it to the network using the connection state setup using the AES key received from the initiator
- Update the connection state to use the new secret key for all future communication
- Inform the application layer about the received connection and what are its capabilities


connect
socket.connect()
socket connected
handshake
create & send Hello
handle Auth Hello
invoke CB
Party
party.connect(
options,
CB
)
create Connection
handle Hello
create session
update sessioncache
create & send Auth Hello
Party
party.listen(
options,
connListener
)
Connected
Connected



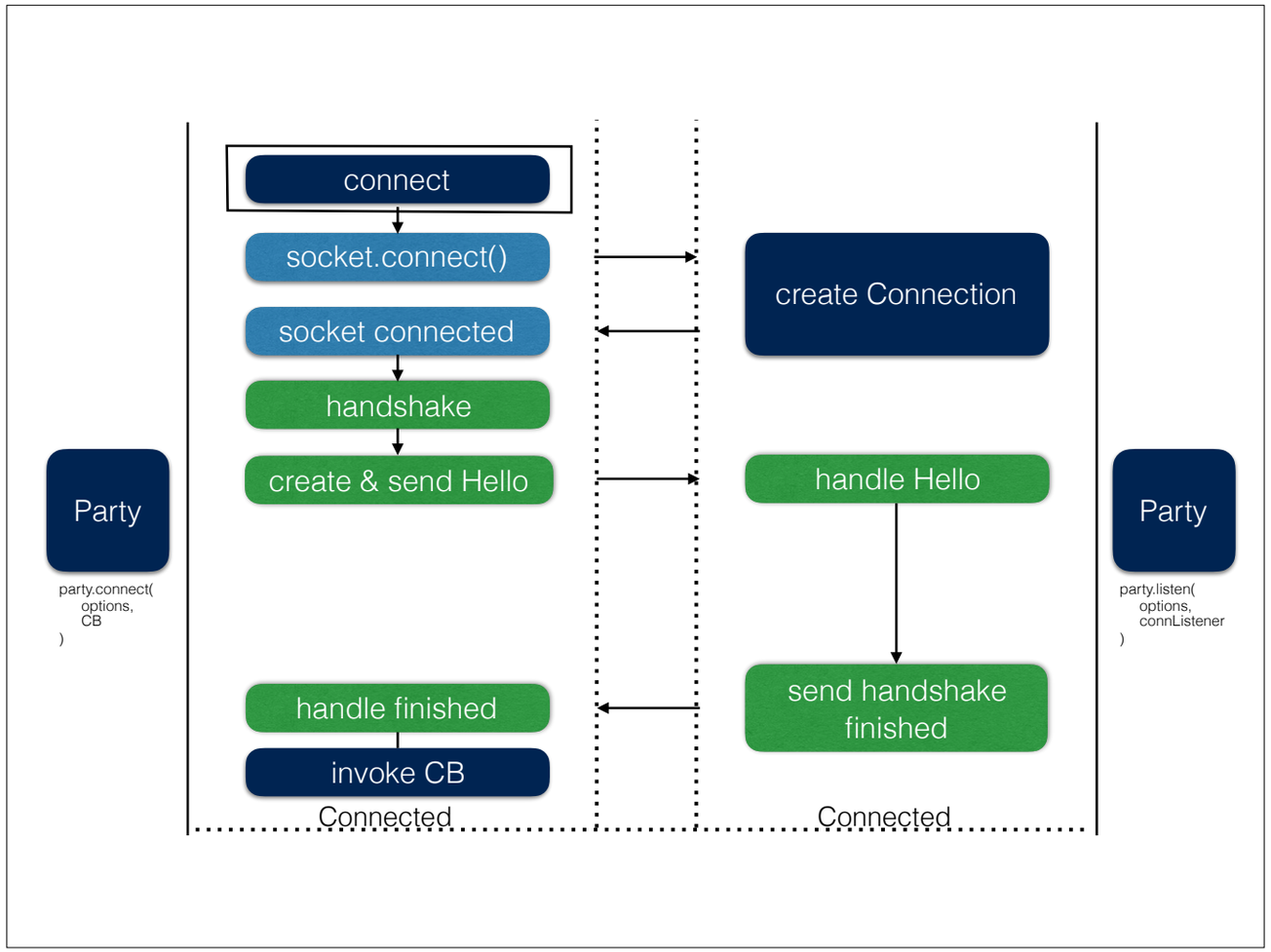
- Read the record header and corresponding seif record
- Decrypt it using the old secret generated by the initiator itself
- unwrap the auth hello record and decrypt the new secret using its own private key
- update the session cache using the newly received session id
- Update the connection state to use the new key for future communication



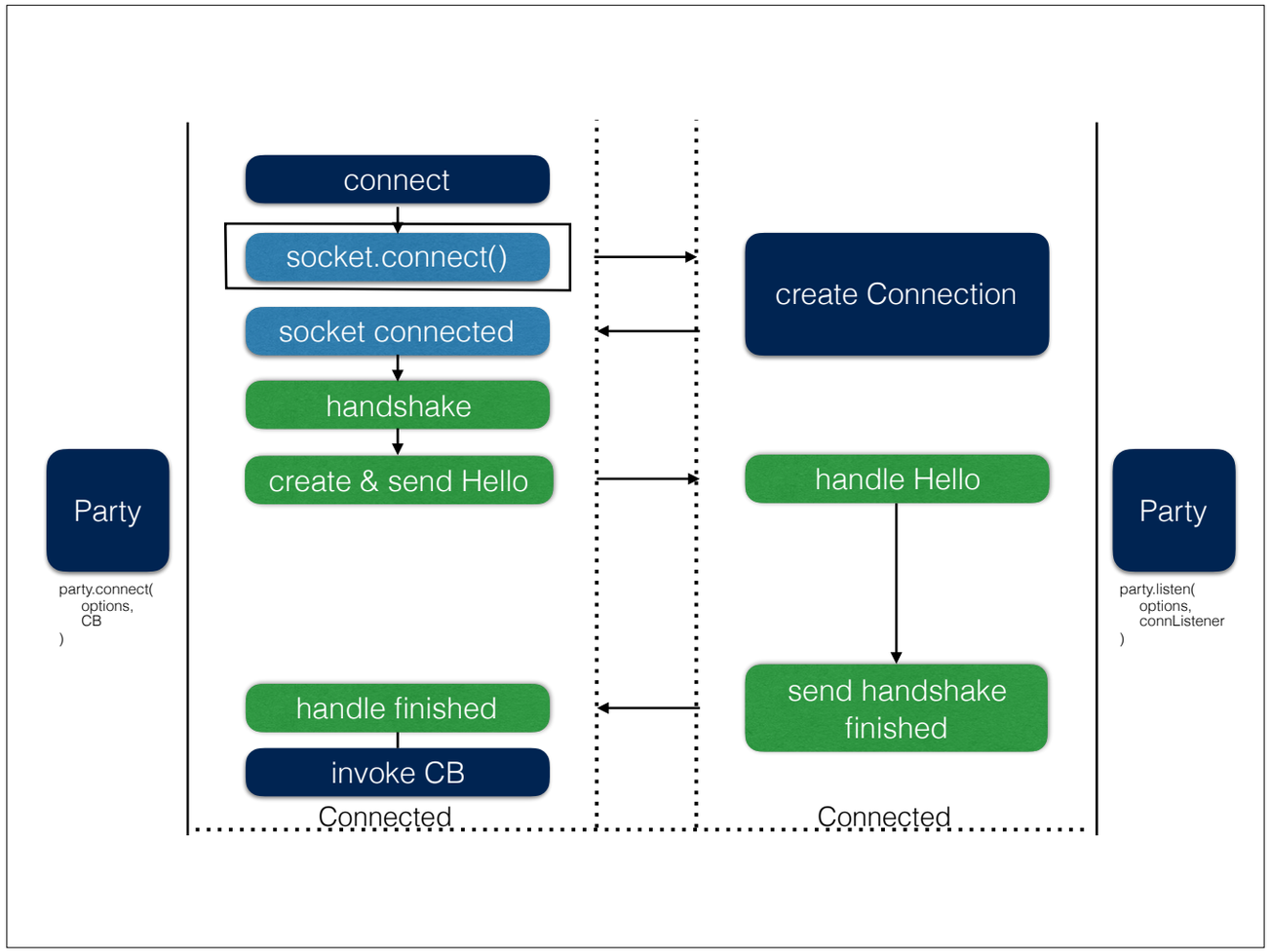
- Connection now established at both ends
- One thing that is intentionally left out is what happens in case of errors - such as validation failures, or encryption failures etc.
- We have a very simple strategy for these cases: the party just breaks the connection without any information. we believe its best to give as less information as possible.

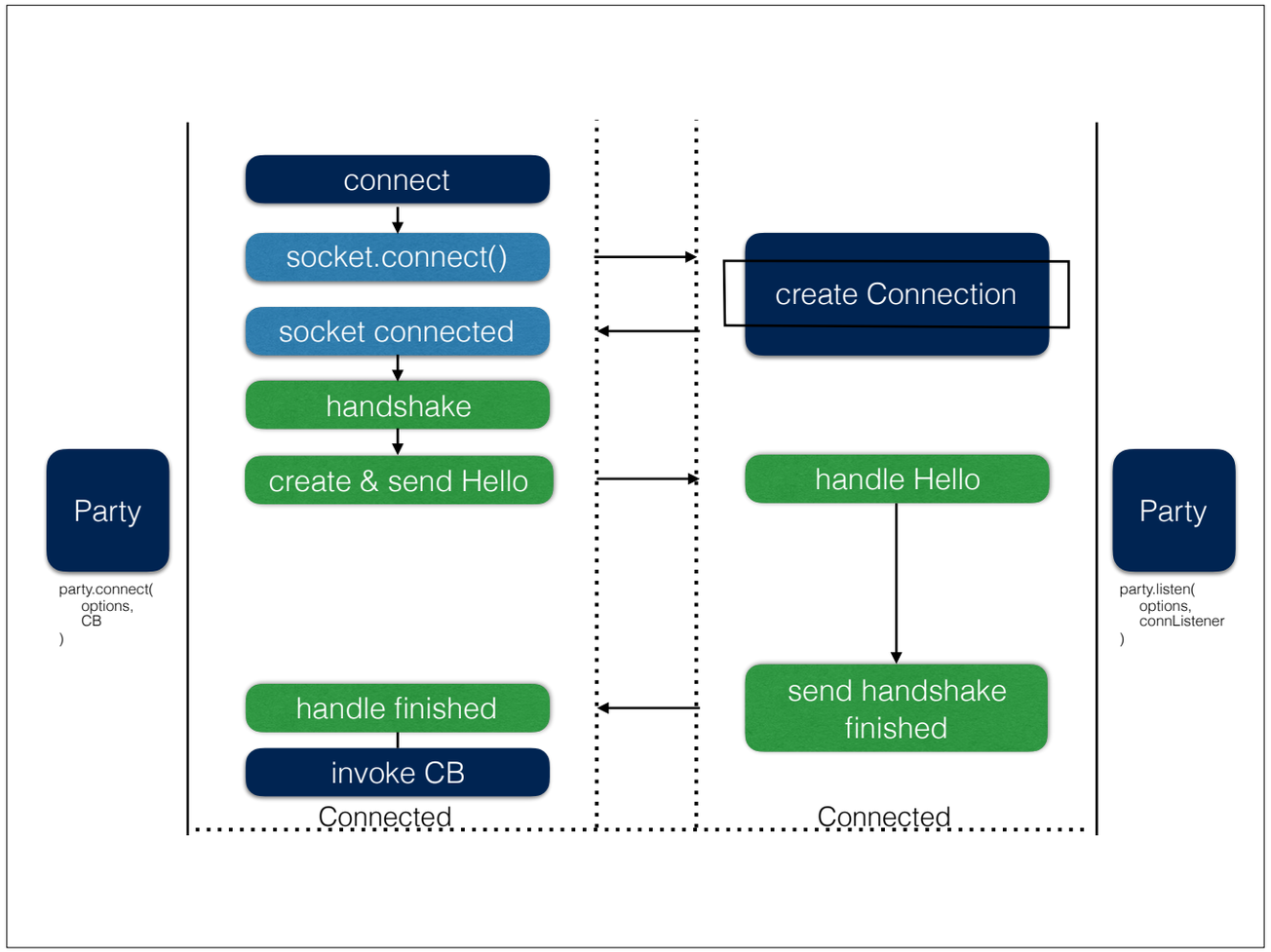
Connection Setup with Sessions

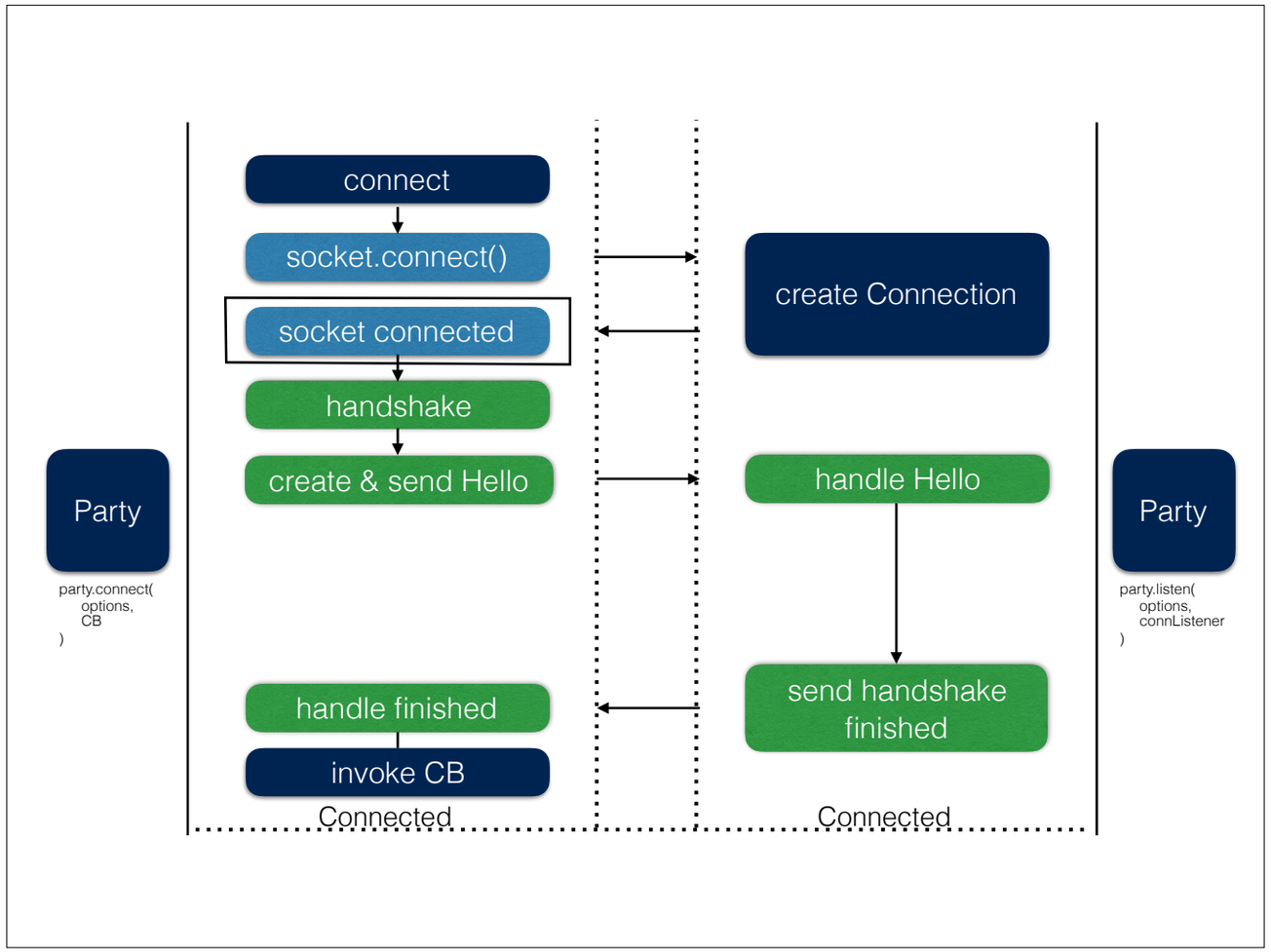
- Lets go over the flow of the connection setup in the case where a prior session exists with the other party

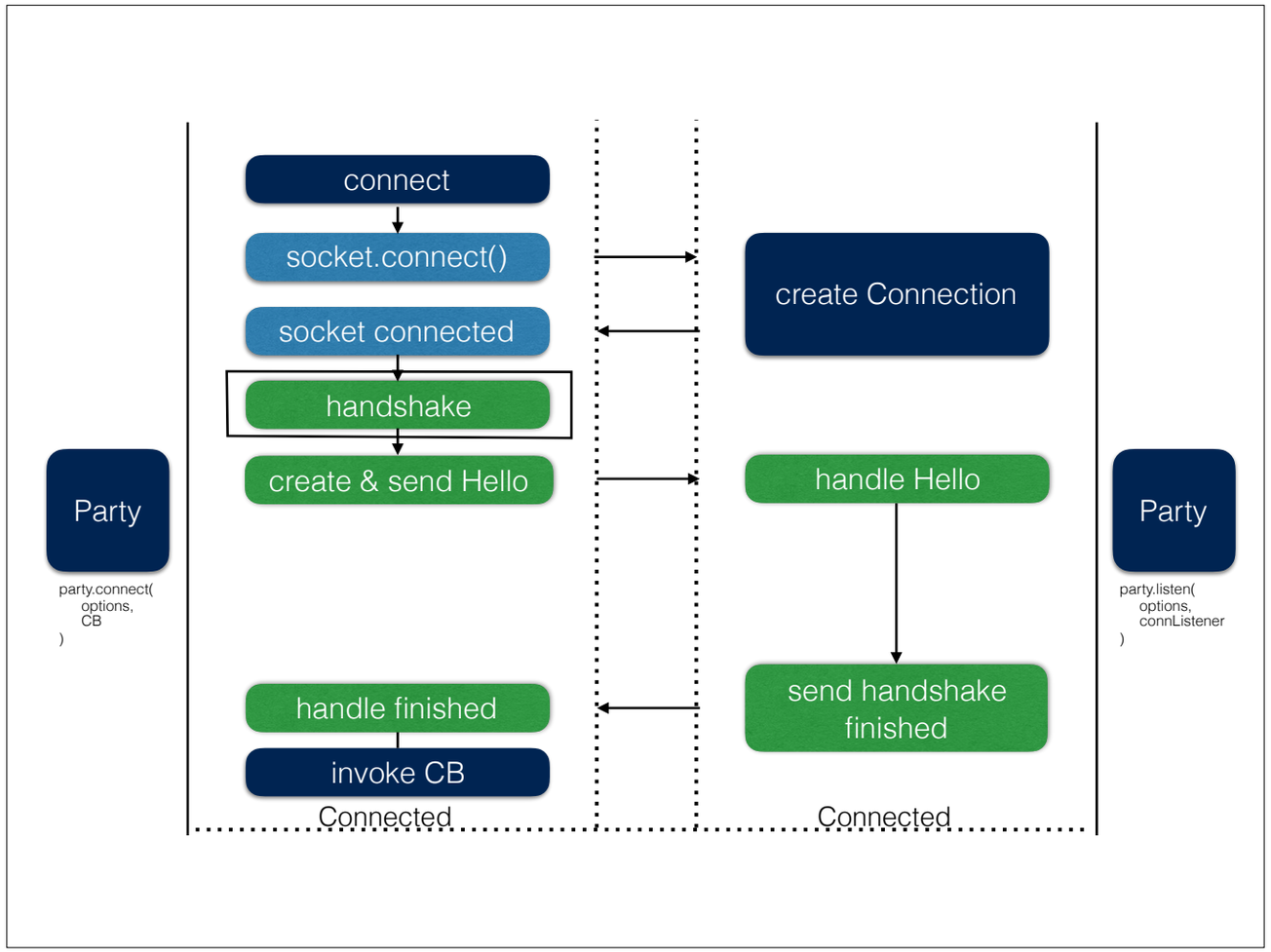


- The flow is similar but way faster and simpler.
- The green blocks are the ones which are modified
- Initial steps are the same
- We setup a TCP connection on both ends

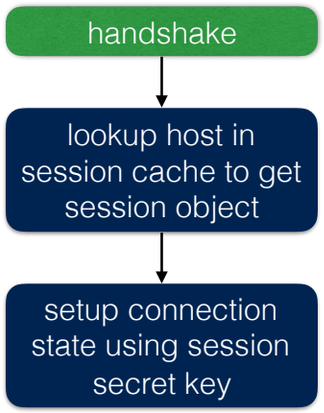

connect
socket.connect()
socket connected
handshake
create & send Hello
handle finished
invoke CB
Connected
create Connection
handle Hello
send handshake finished
Connected
Party
party.connect(
options,
CB
)
Party
party.listen(
options,
connListener
)


connect
socket.connect()
socket connected
handshake
create & send Hello
handle finished
invoke CB
Connected
create Connection
handle Hello
send handshake finished
Connected
Party
party.connect(
options,
CB
)
Party
party.listen(
options,
connListener
)

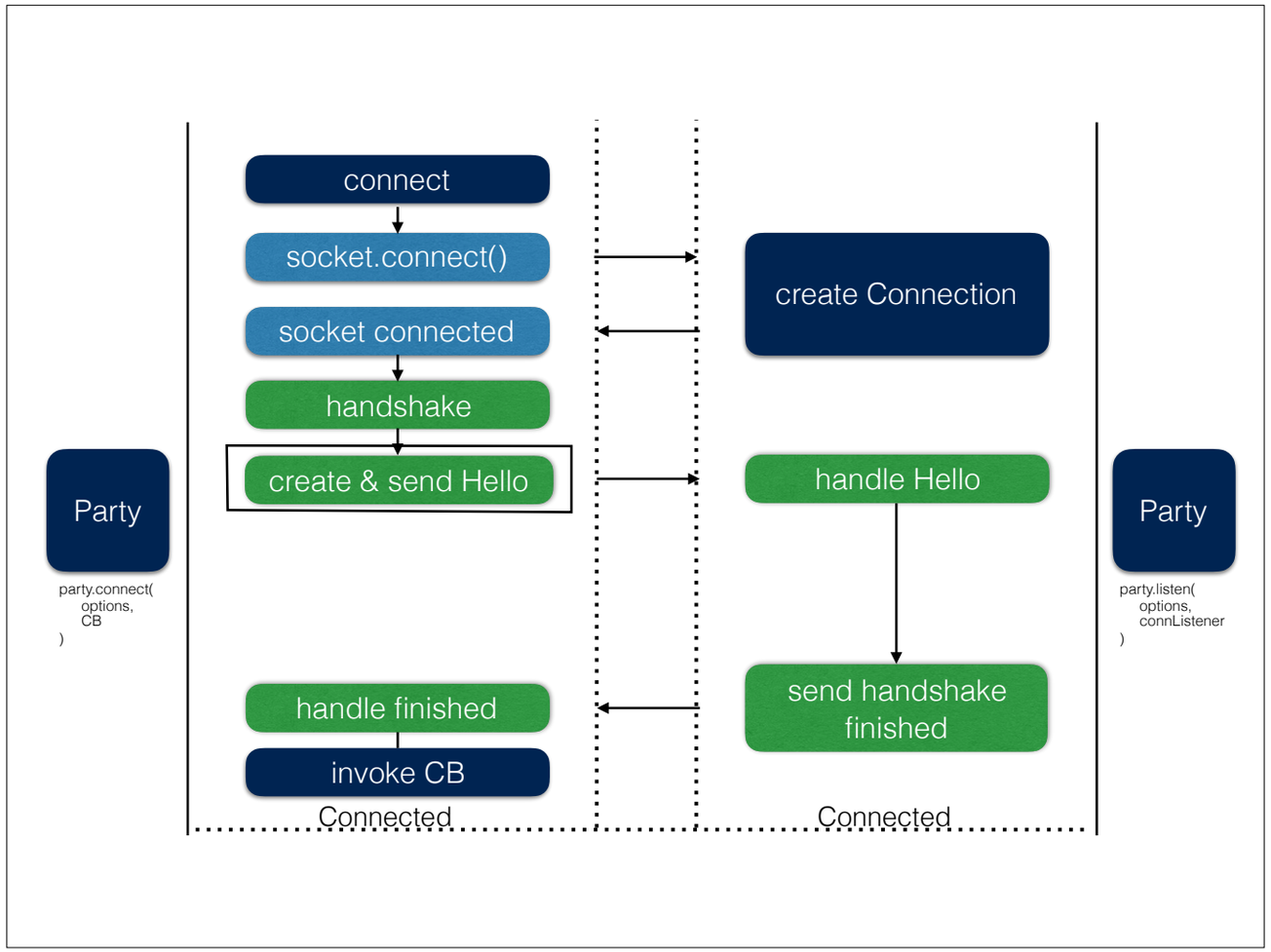

connect
socket.connect()
socket connected
handshake
create & send Hello
handle finished
invoke CB
Connected
create Connection
handle Hello
send handshake finished
Connected
Party
party.connect(
options,
CB
)
Party
party.listen(
options,
connListener
)

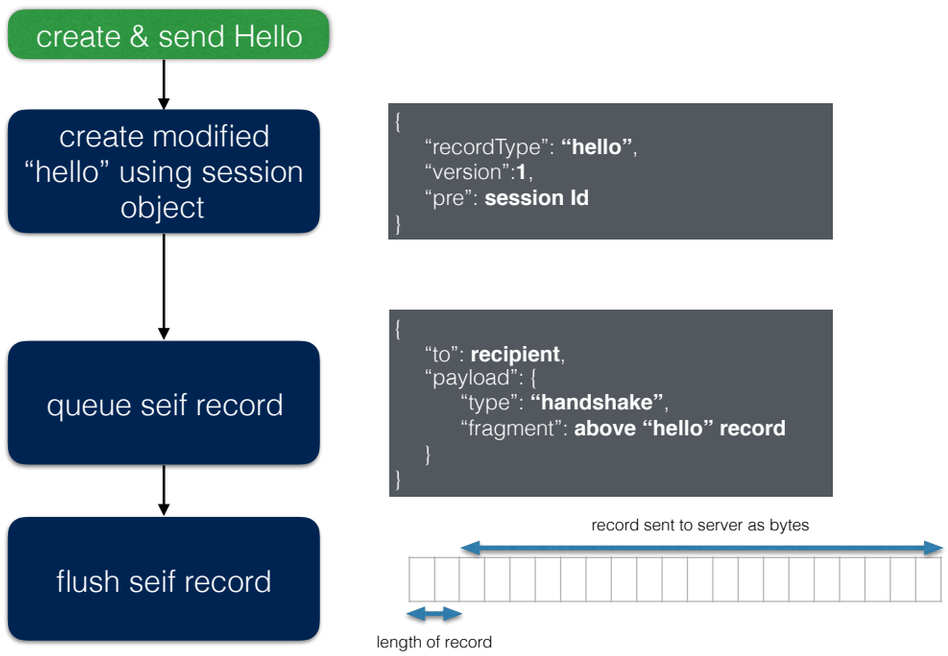


- Ready to perform the handshake process

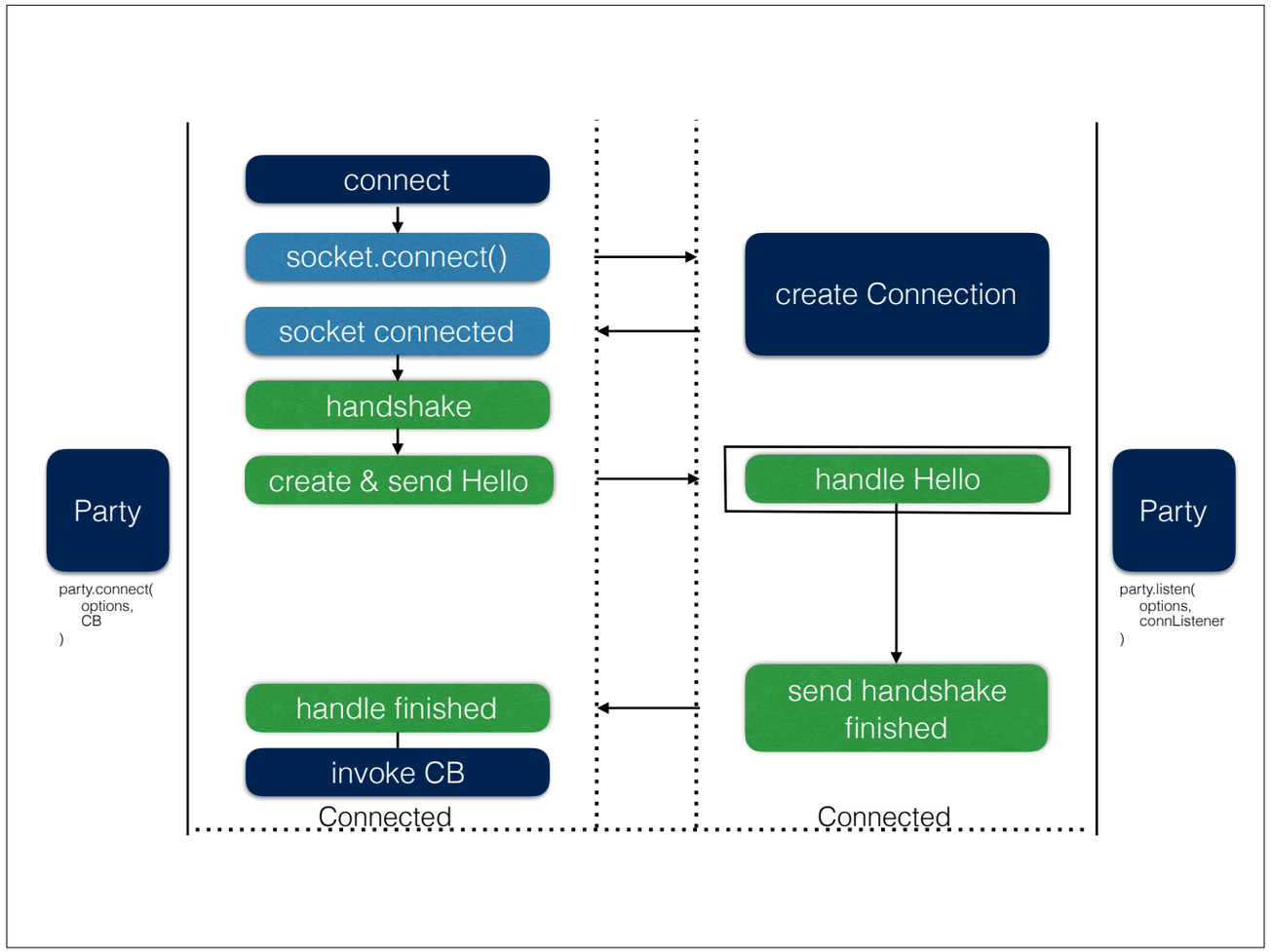


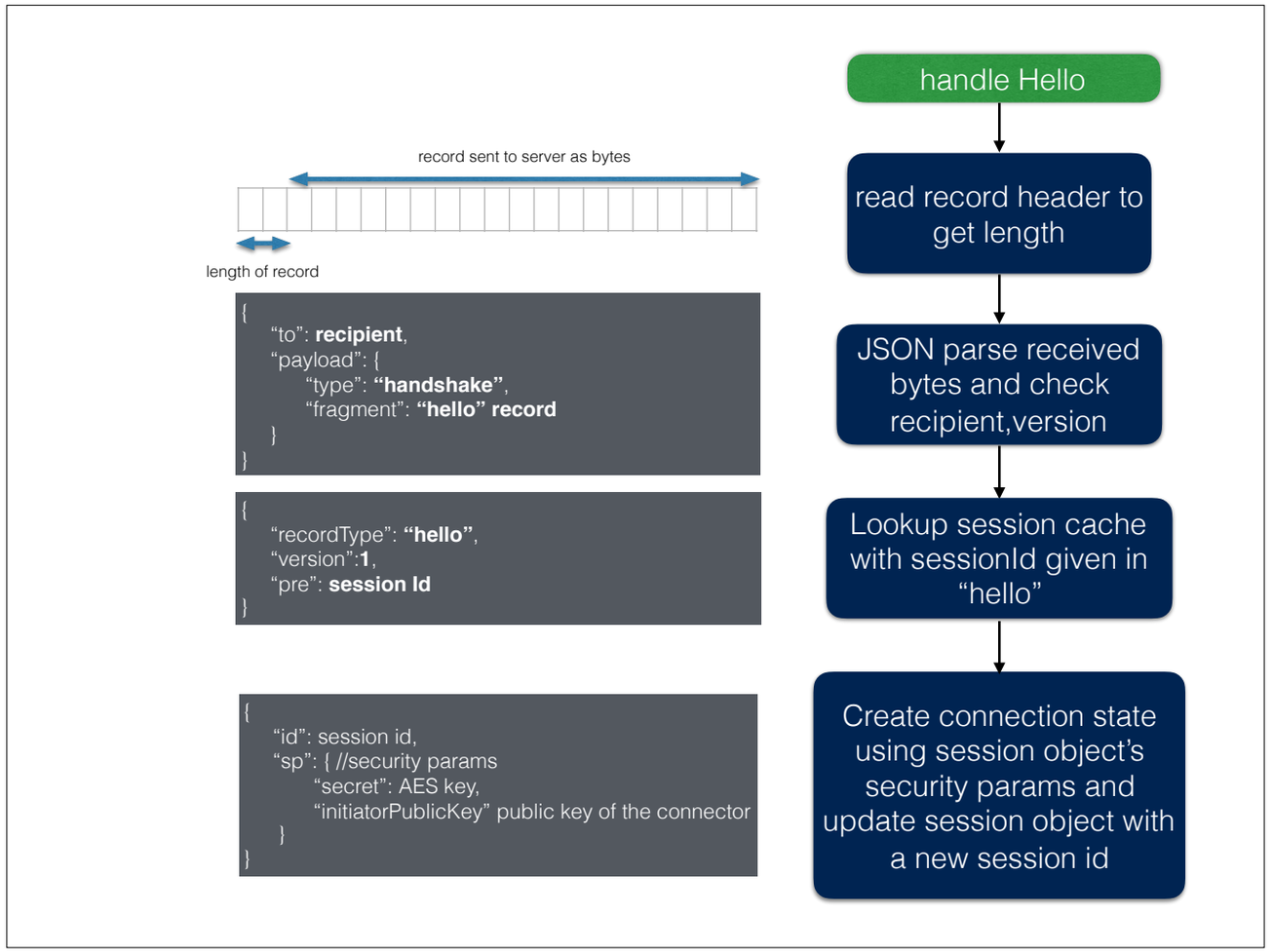
- This time, the party first looks up the session cache to check if a corresponding session exists
- Assuming it does, it creates a pending connection state using the secret key obtained from the session


connect
socket.connect()
socket connected
handshake
create & send Hello
handle finished
invoke CB
Connected
create Connection
handle Hello
send handshake finished
Connected
Party
party.connect(
options,
CB
)
Party
party.listen(
options,
connListener
)

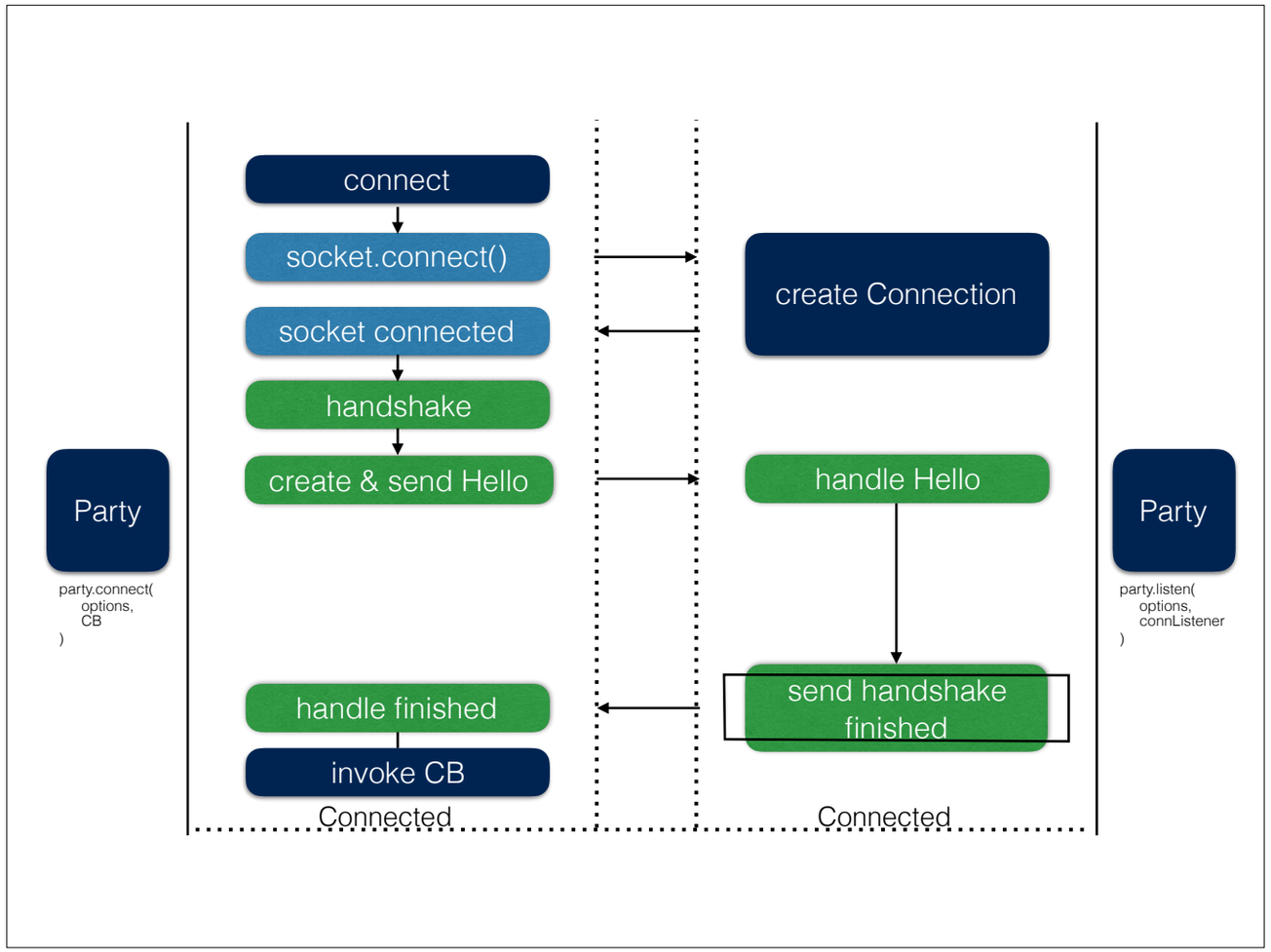


- Next it creates a modified hello message
- This message only includes the session id
- Wrapped seif Record is then flushed in the clear

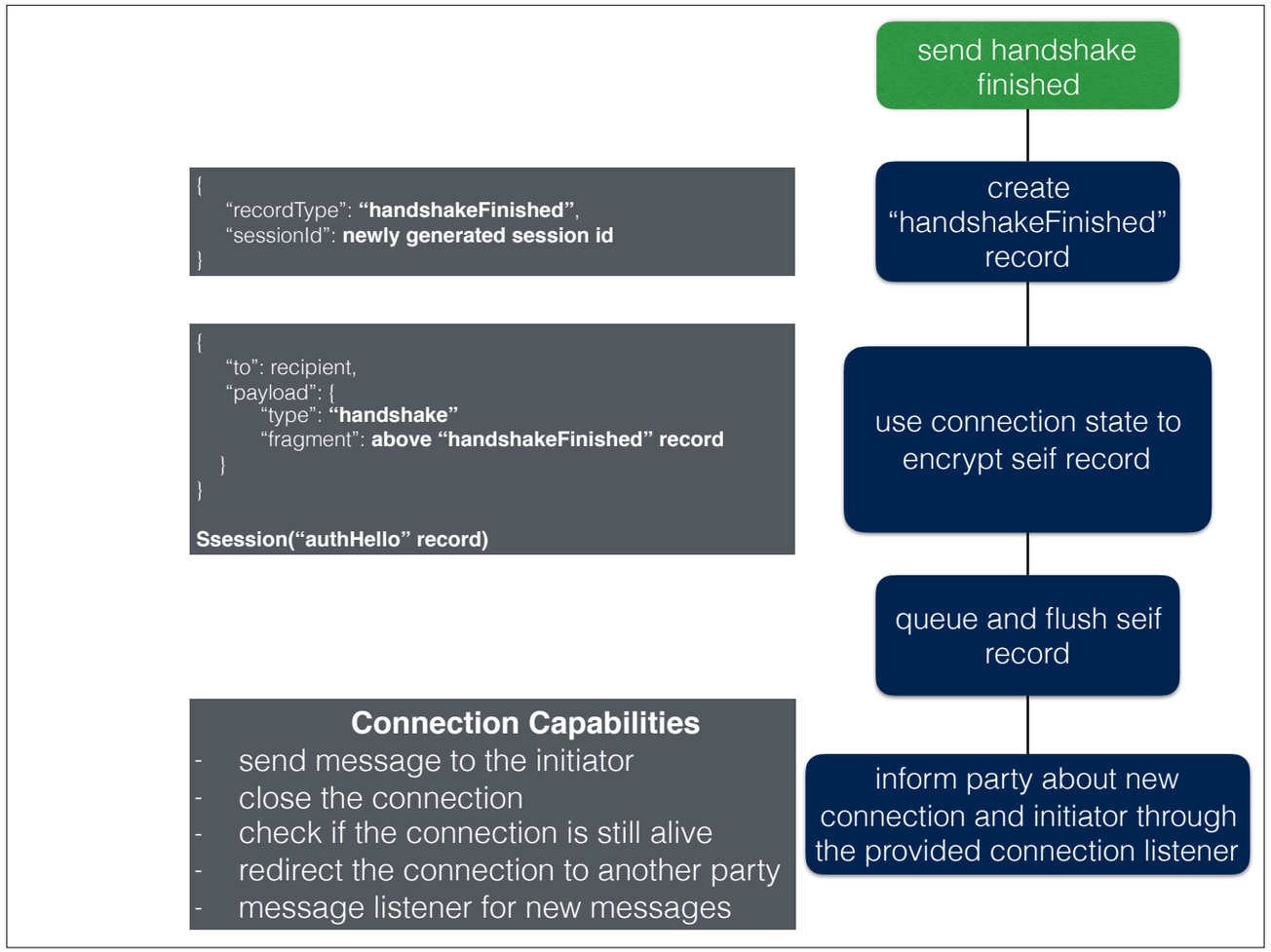

connect
socket.connect()
socket connected
handshake
create & send Hello
handle finished
invoke CB
Party
party.connect(
options,
CB
)
create Connection
handle Hello
send handshake
finished
Party
party.listen(
options,
connListener
)
Connected
Connected



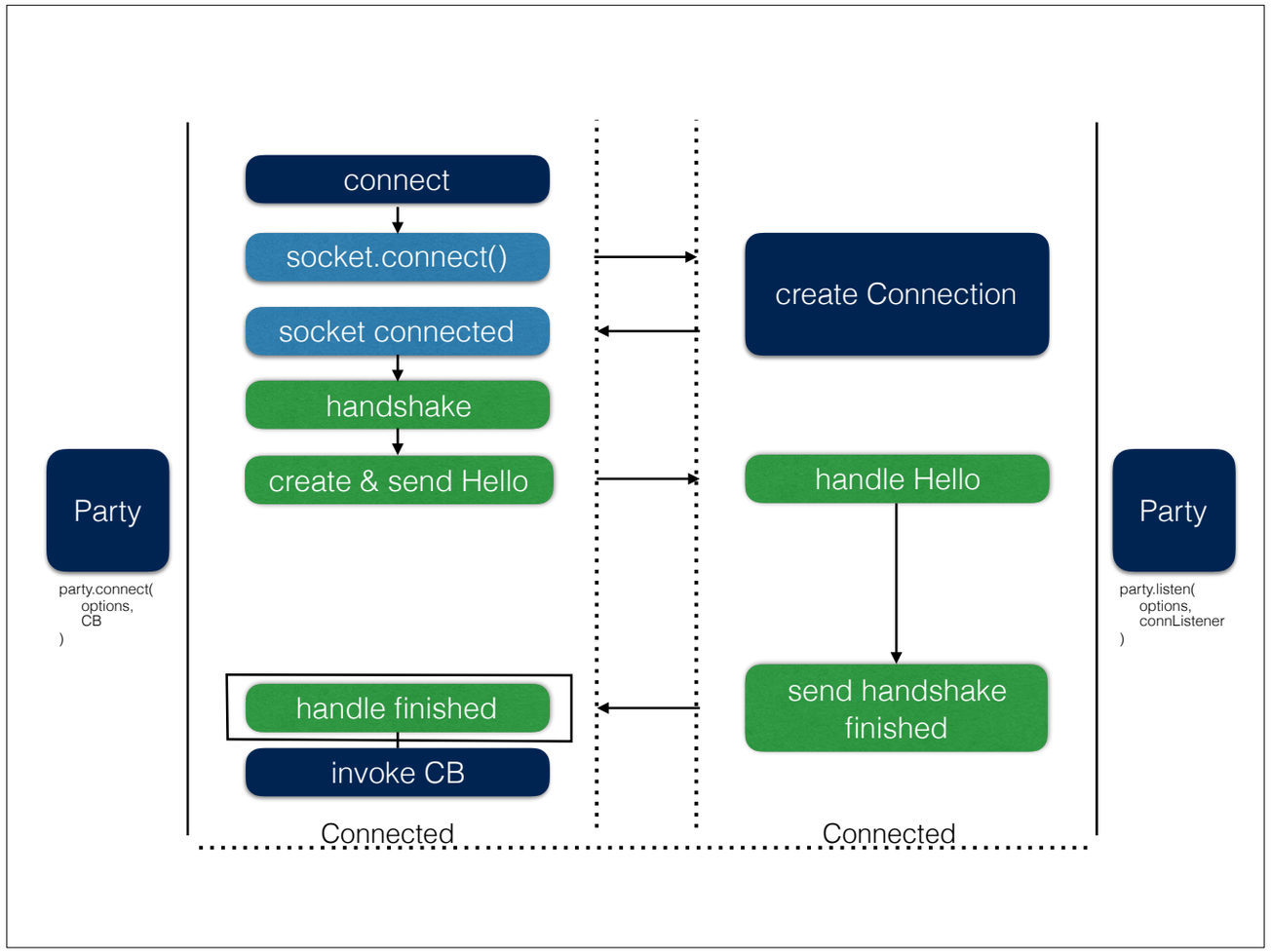
- The listening party receives the hello and unwraps it to get the session id
- Assuming it is able to find this session id in its cache, the party now can setup its connection state using the secret key stored in the session
- The party also generates a new session id, since session ids are defined by the protocol to be for one time use only
- Update the session cache to reflect this new id



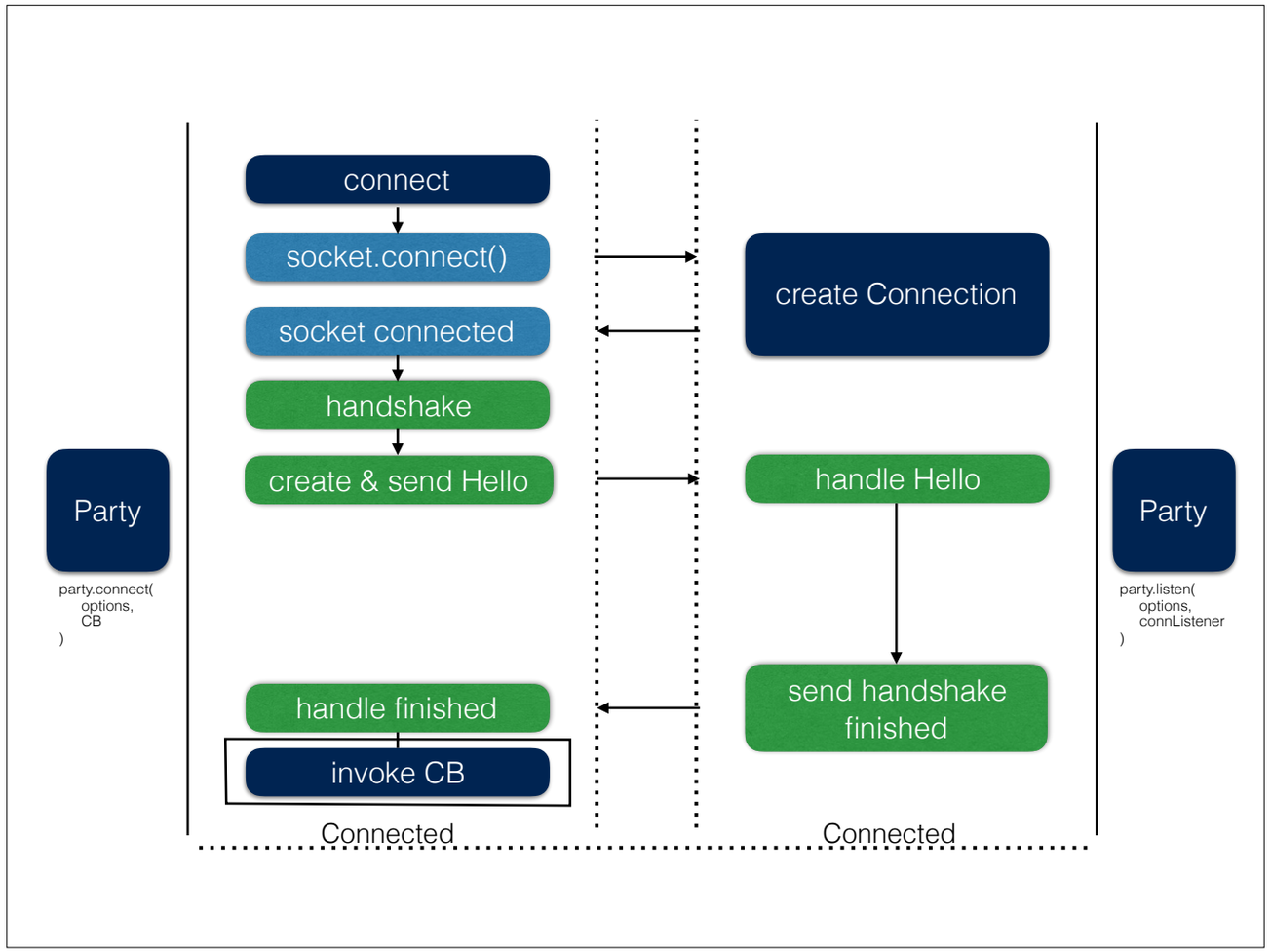
- No need to send an auth hello message



- The party is going to send a finished message which just includes the newly generated session id to be used for the next time a new connection is attempted
- This is encrypted using the connection state created using the secret obtained from the prior session
- As in the previous case, the party informs the application about the new connection and its capabilities

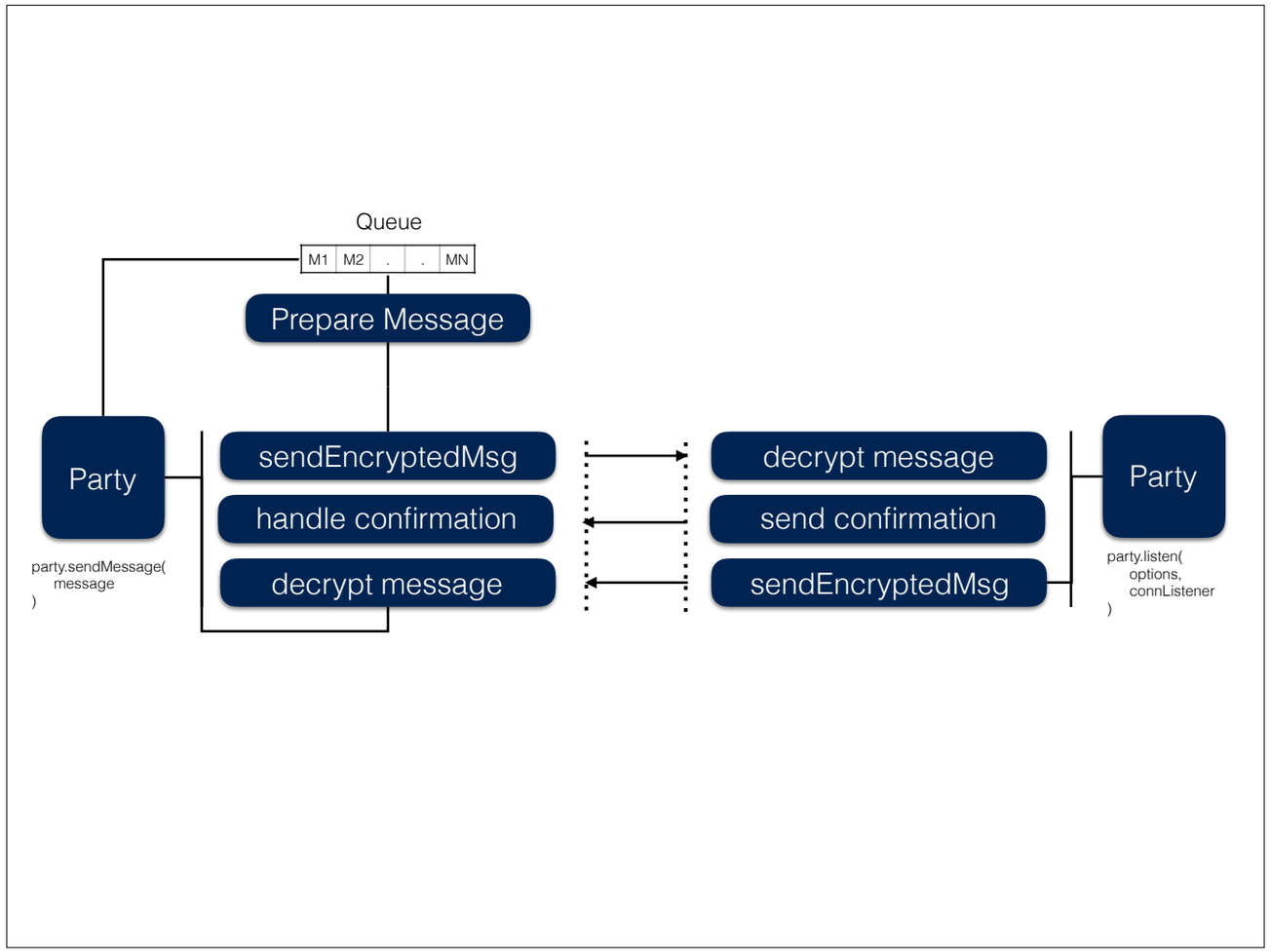


- The initiator receives, decrypts the finished message using the secret from the prior session
- It updates the session cache to use the new one time use session id

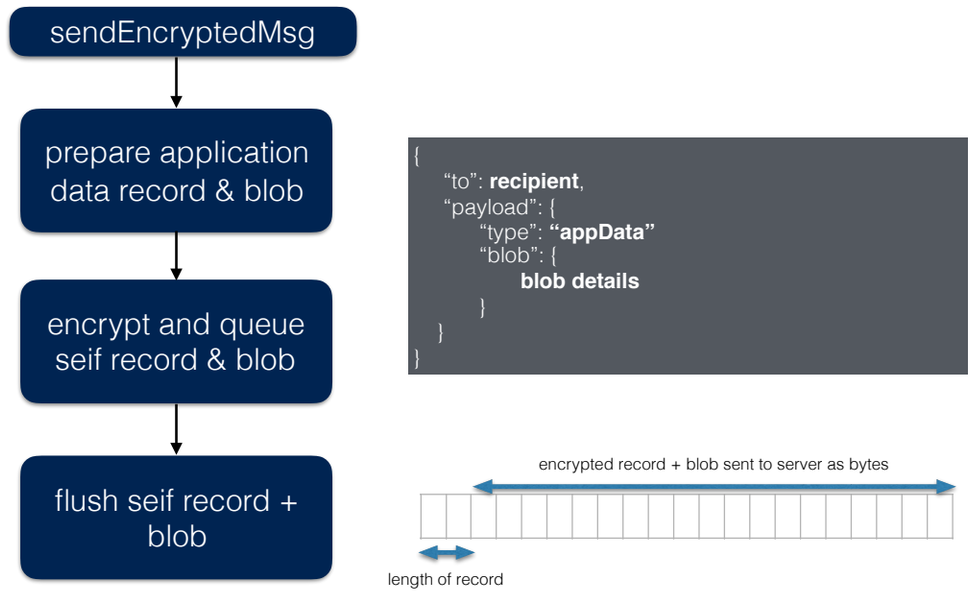


- Connection is now established
- All future application messages will be encrypted using this same secret

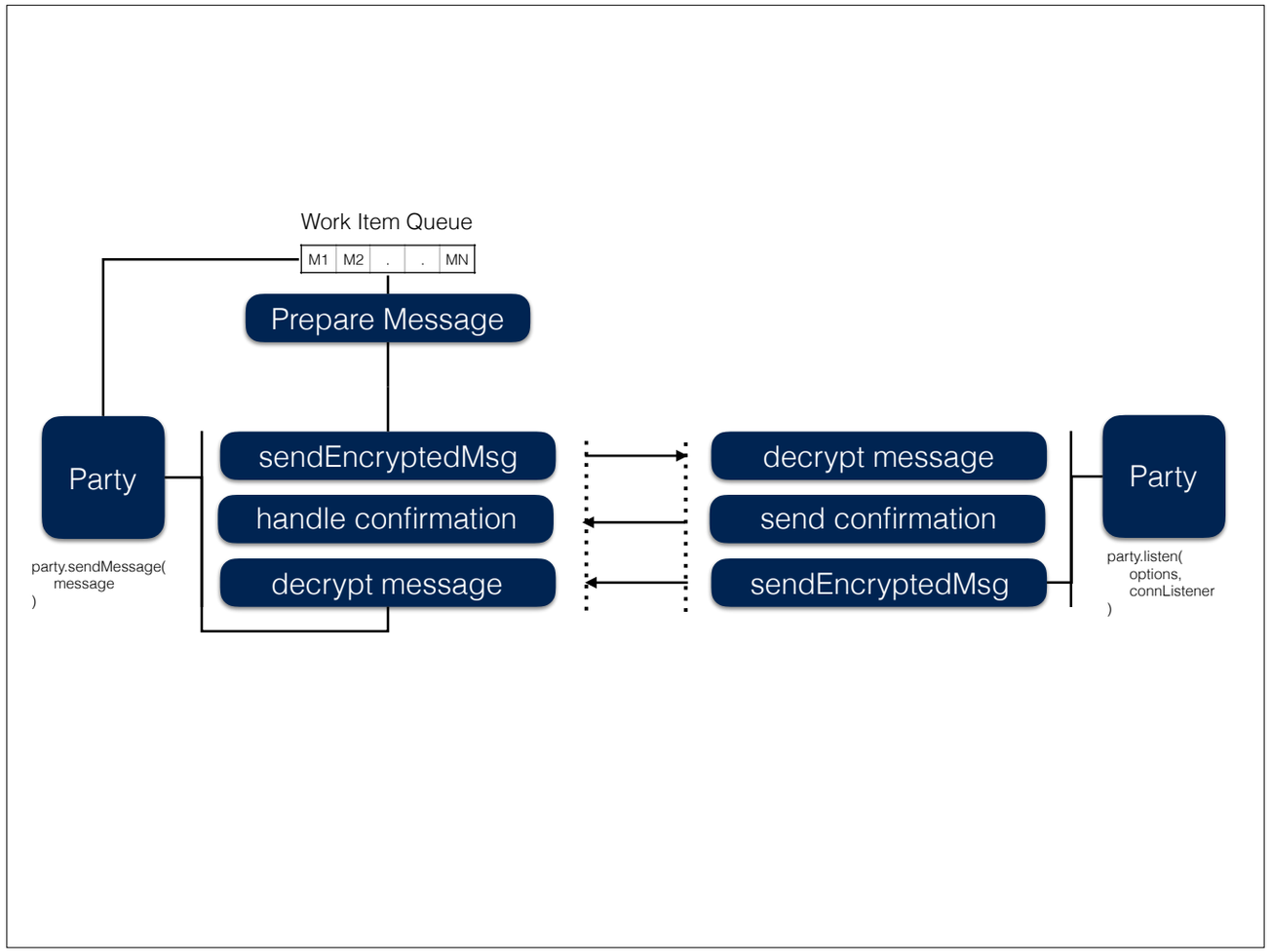
Sending a message



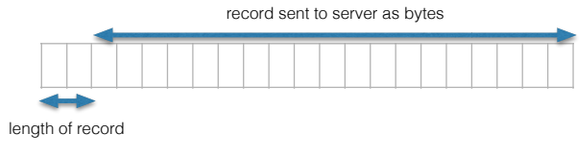
- All future messages are encrypted using the negotiated secret key
- Messages are queued with an associated id



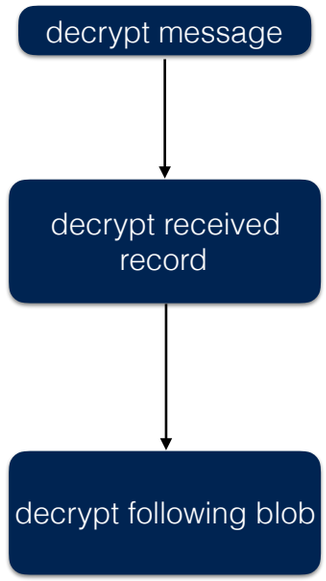
- Prepare the application data record which stores the length of the message sent as a blob
- Encrypt the seif record and the blob using the shared secret key and flush it to the network
- So in this case, the length header refers to the length of the seif record and the seif record in turn has the length of the following blob

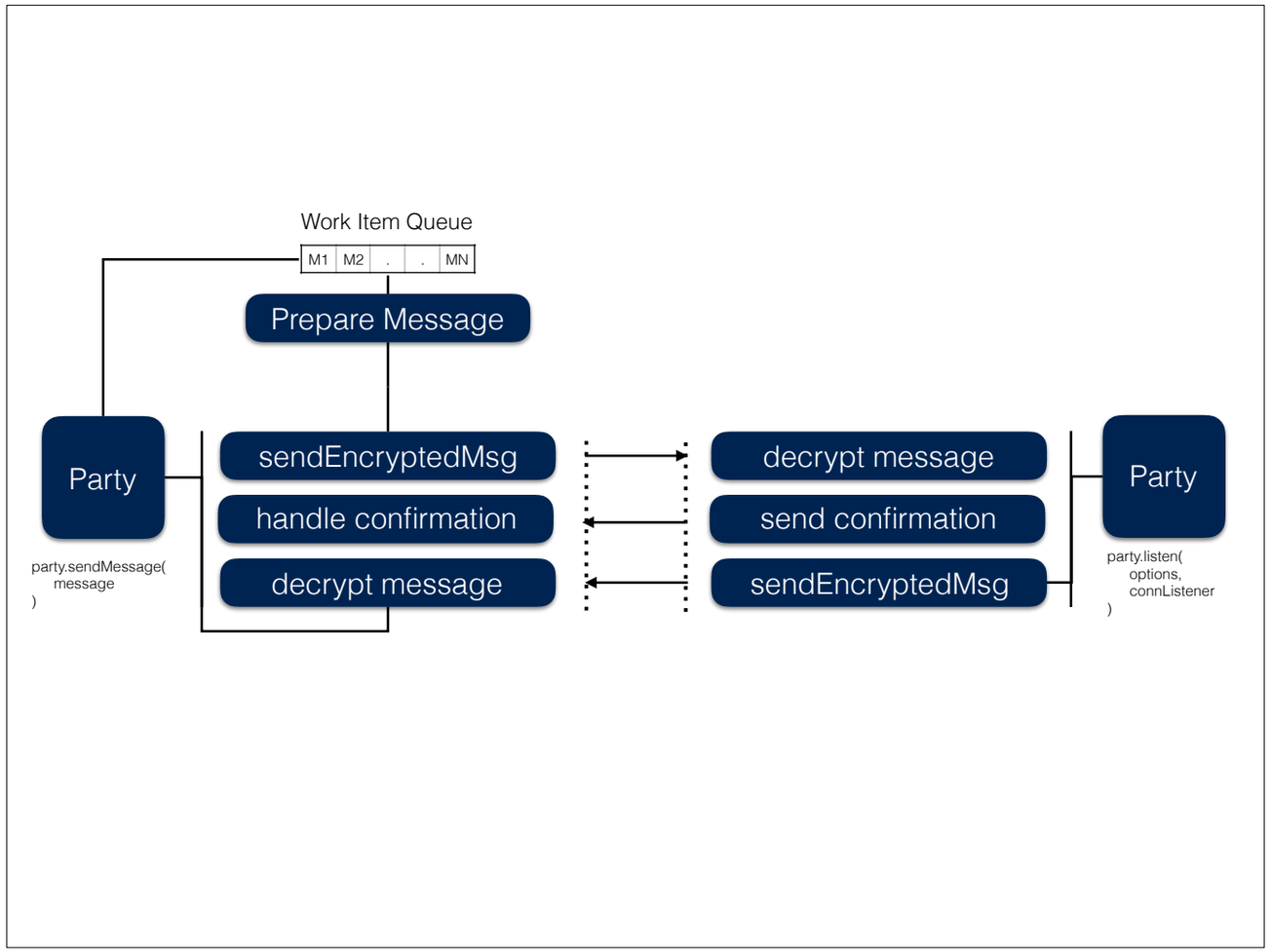


- Server sends a confirmation record with the id of the confirmed message
- Client receives the confirmation and deletes the relevant message from its queue




```
{
    “to”: recipient,
    “payload”: {
        “type”: “appData”
        “blob”: {
            blob details
        }
    }
}
```



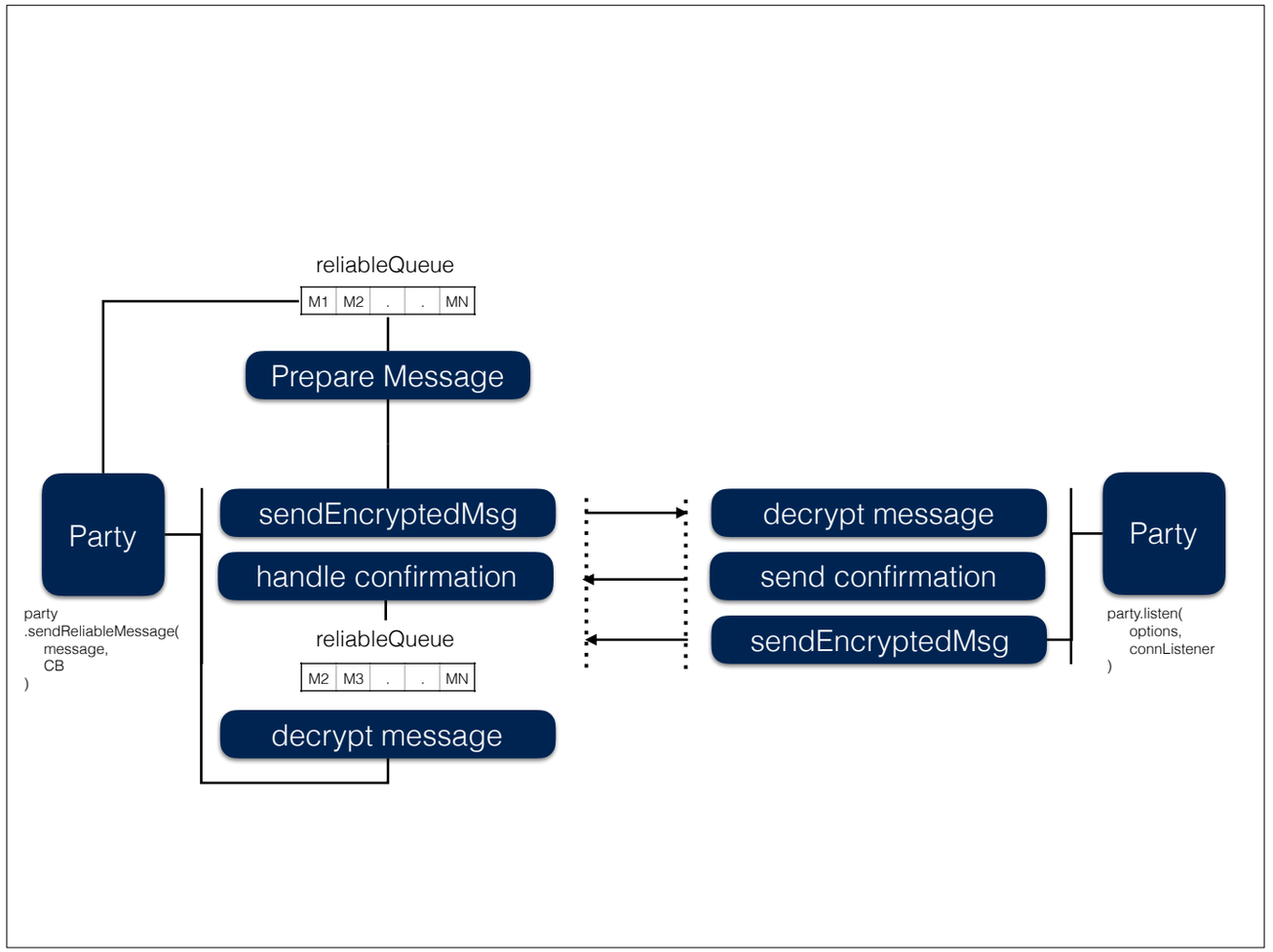


- Once message is decrypted, the party sends a confirmation including the message id back to the sender
- The application layer is then informed about a new received message
- The sender receives the confirmation and removes the message from the queue

Messages

- sendMessage
- sendUnreliableMessage
- sendReliableMessage

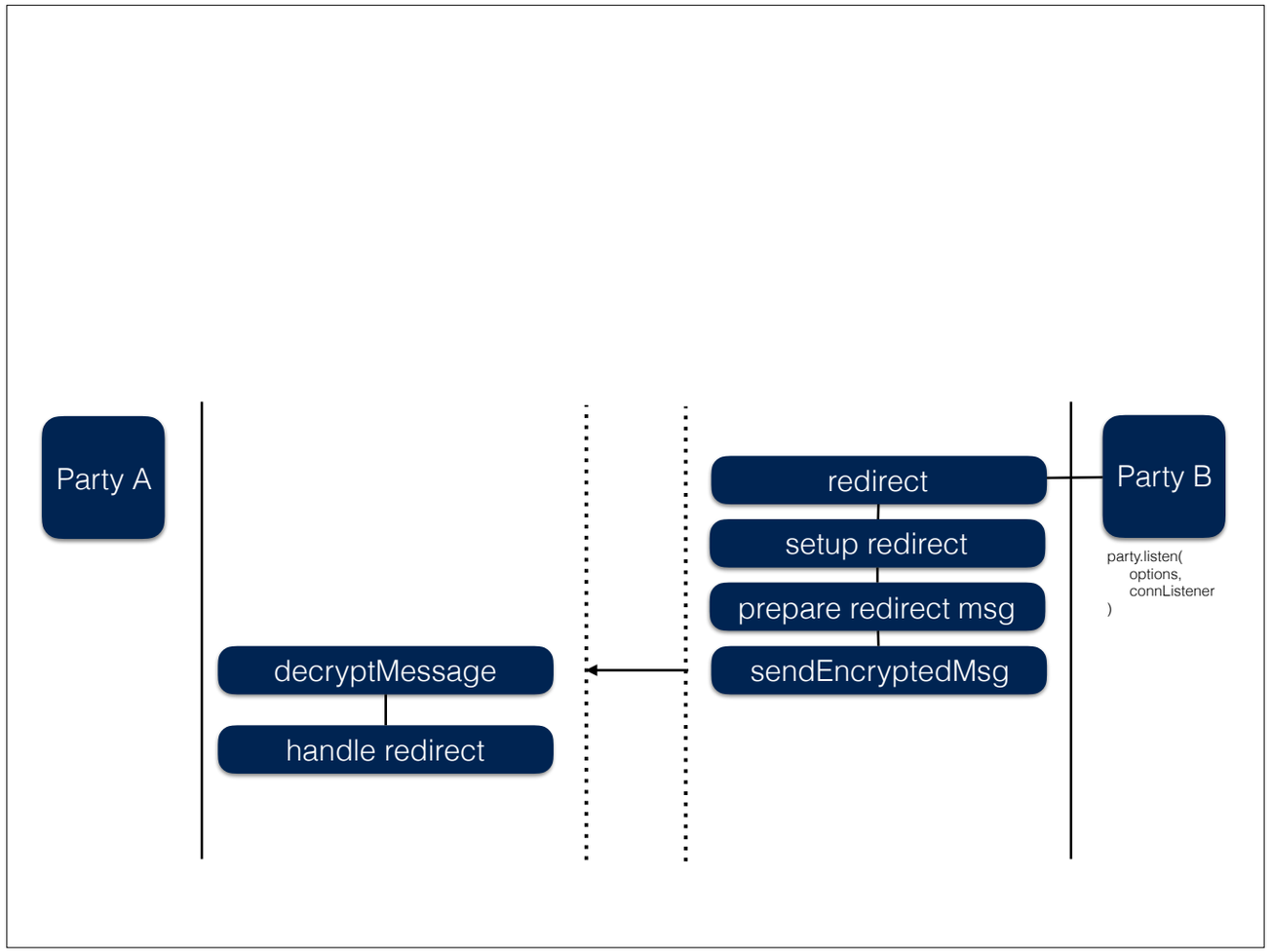
- Unreliable message - no required confirmation, we do not need to queue the message at all
- Examples - log, status messages
- Reliable message - guaranteed to be delivered across connection failures



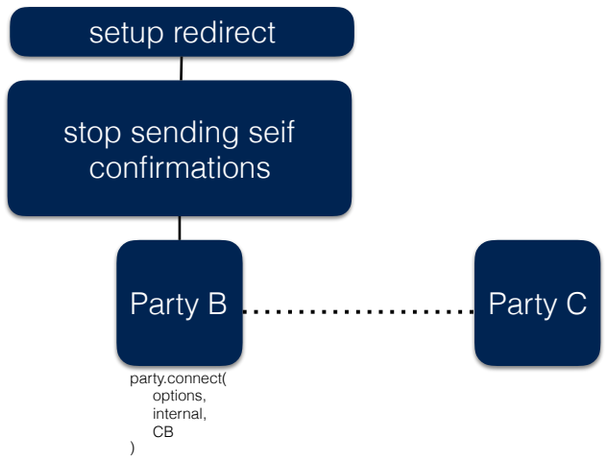
- Reliable message flow
- We need a persistent queue from the application.
- Reliable messages are pushed into the reliable queue. In case of successful confirmation, the message is deleted from the queue.
- In case of failure, we try to send the message the next time a connection is established.

Temporary Redirects

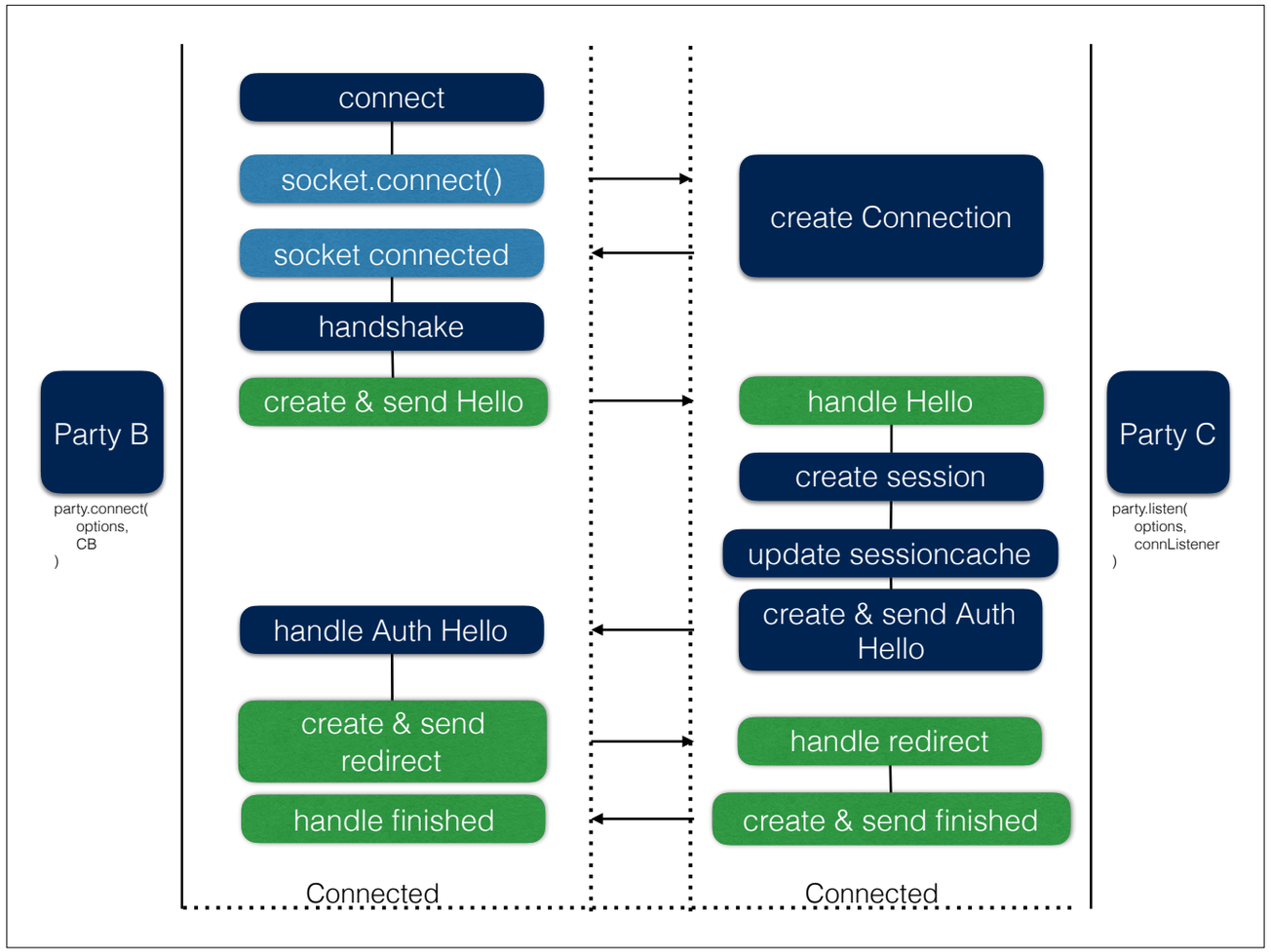
- Two types of redirects
- Temp redirect:



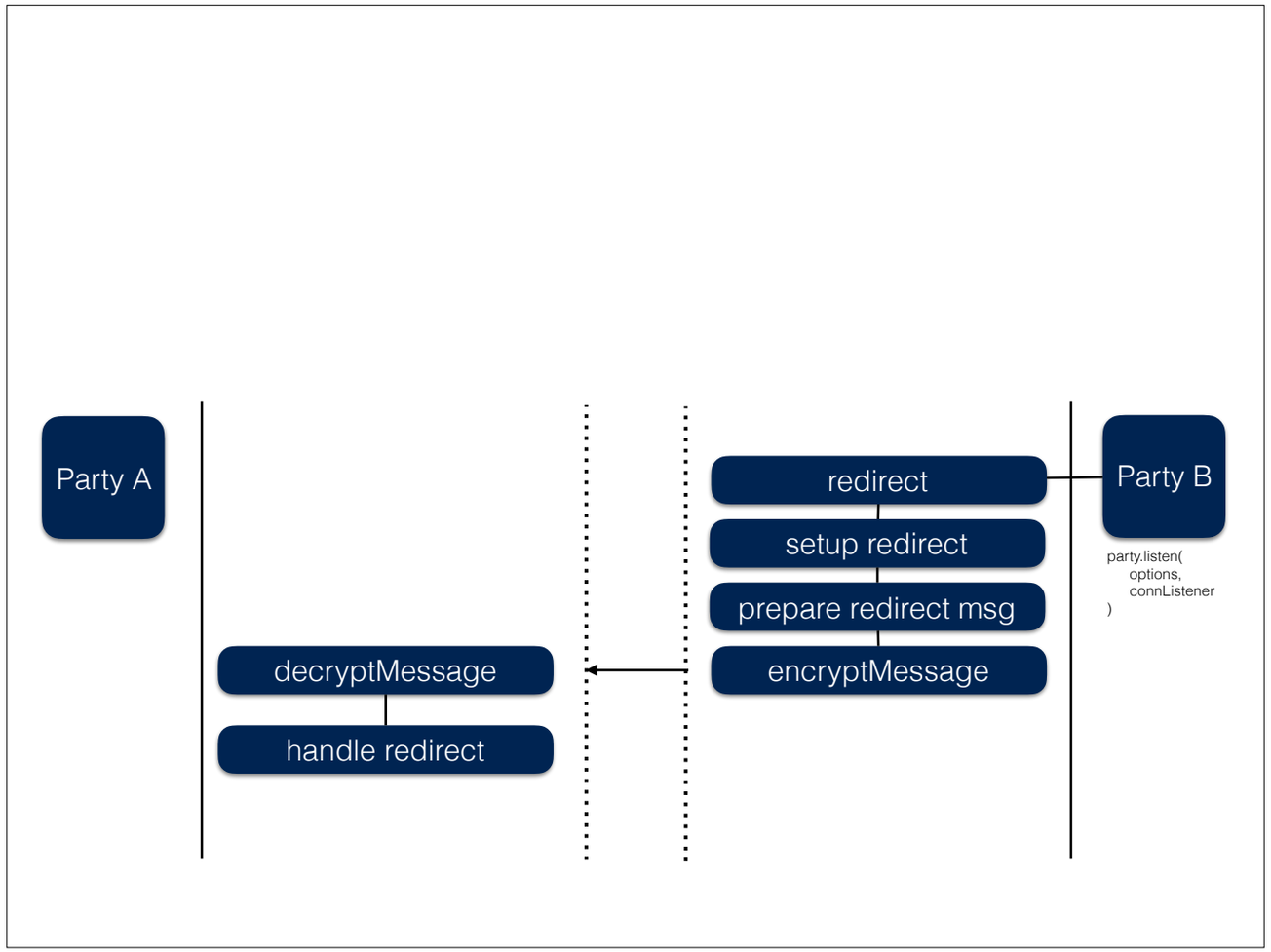
- Redirect a connection for the duration of a session

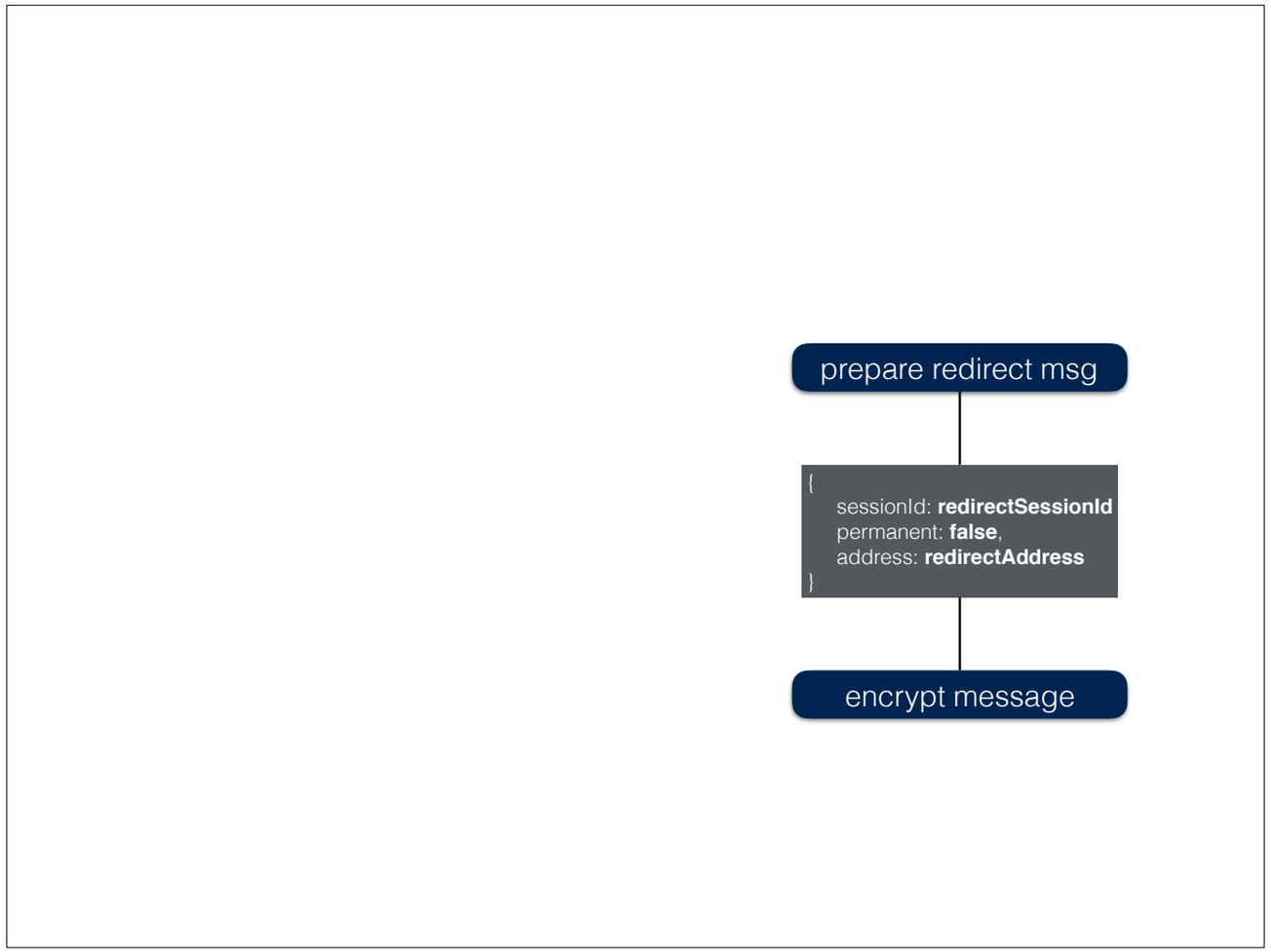


- Stop sending all confirmations or send an indication of a redirect
- Special handshake between party B and C

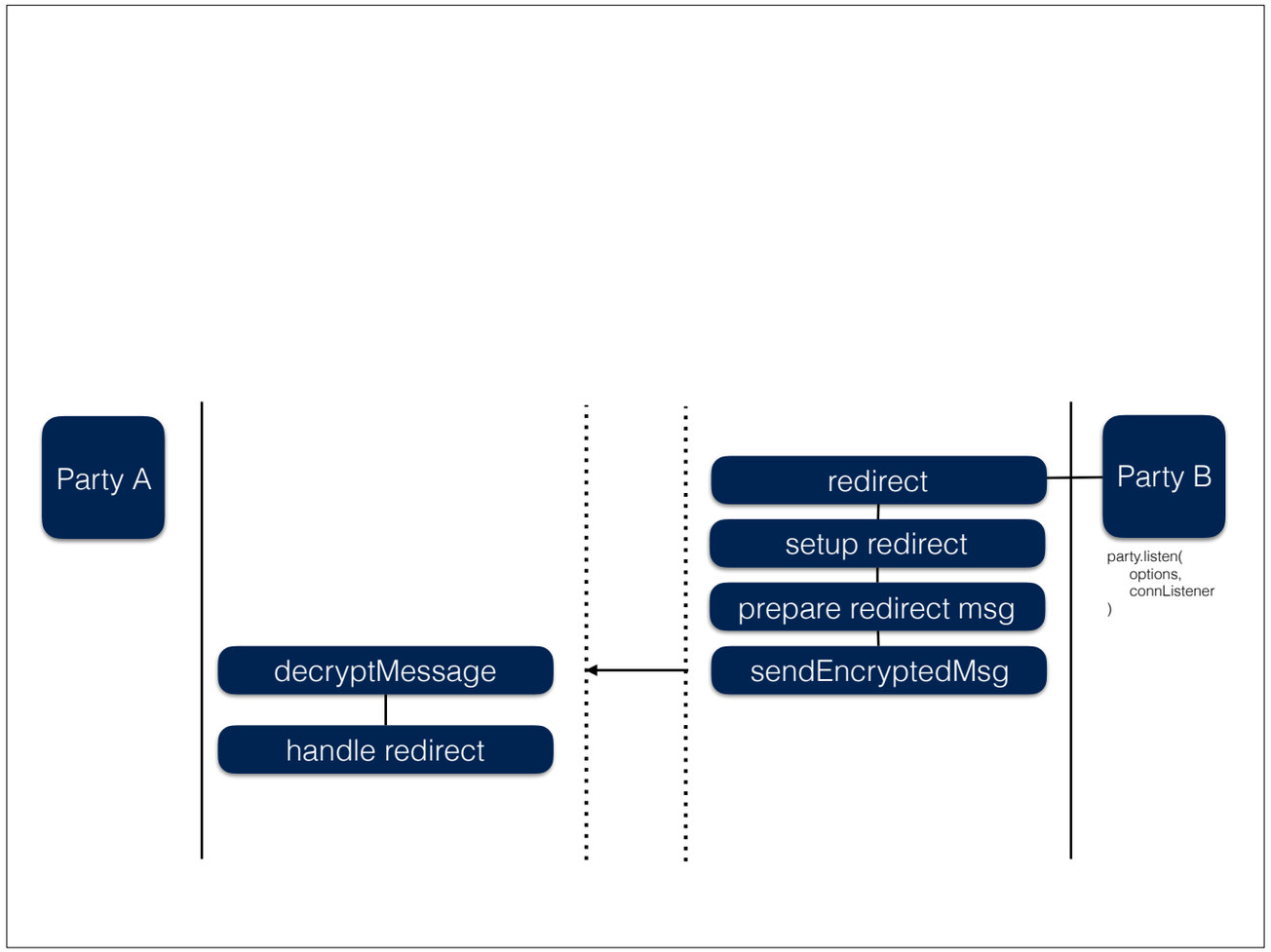


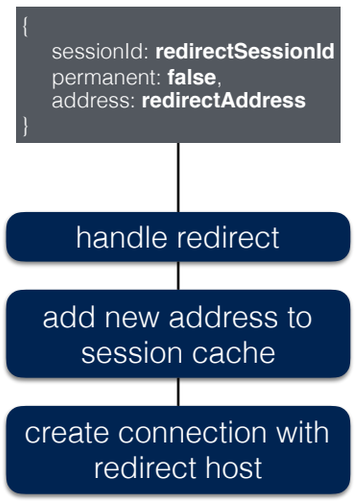
- This flow assumes that no prior session exists between the two parties. If it does exist, then no initiator authentication is required.
- Modified hello message includes an indication of redirect
- The server party now creates a session in the same way and expects a special redirect message encrypted using the shared secret key
- The initiator now sends the session to be redirected in a special handshake message
- The server receives it, and updates its session cache. As before a new one time use session id is generated and sent in a finished message

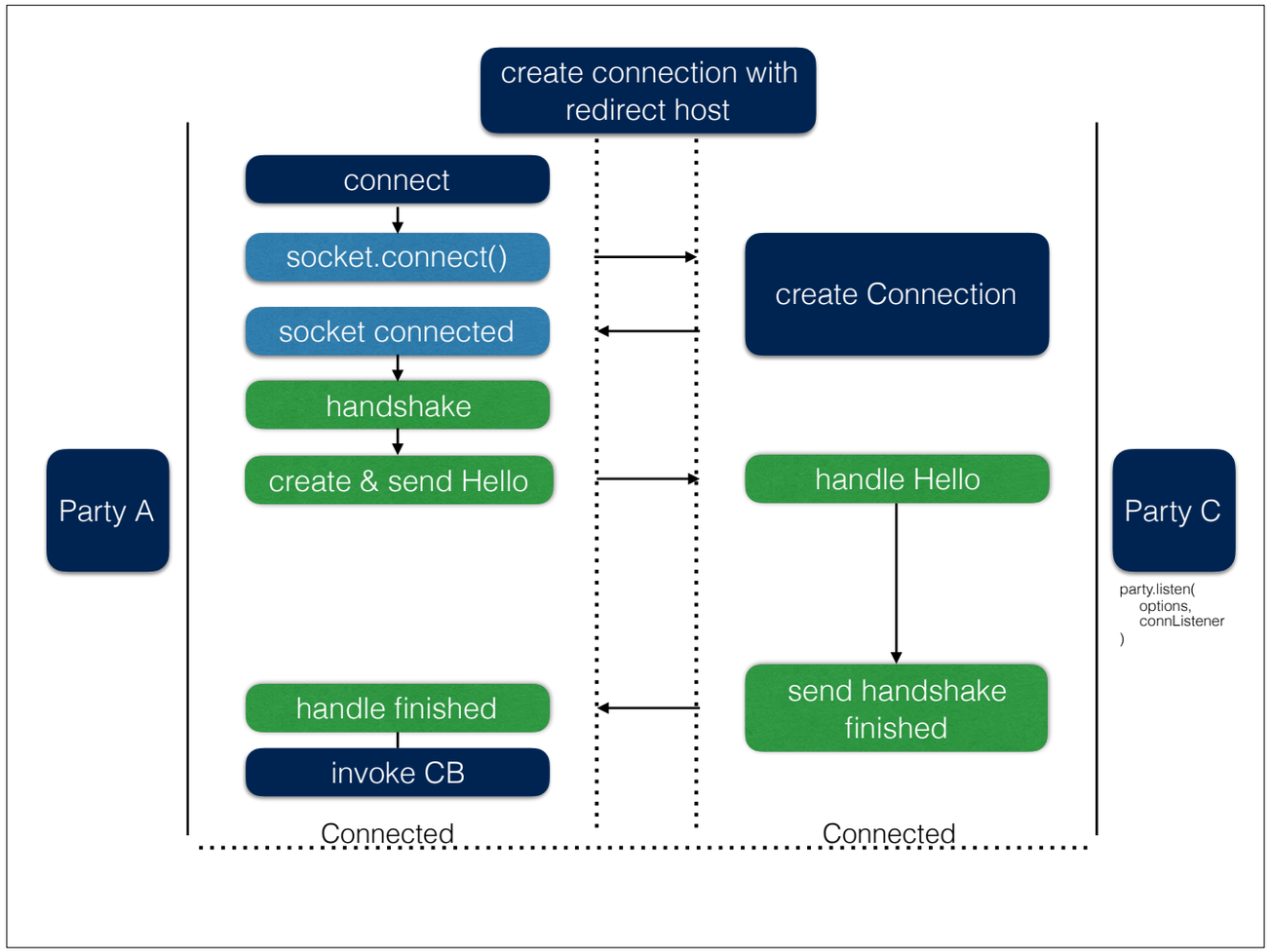

Party A
decryptMessage
handle redirect
redirect
setup redirect
prepare redirect msg
encryptMessage
Party B
party.listen(
options,
connListener
)



- Includes new redirected session id and the address of the new party


Party A
decryptMessage
handle redirect
redirect
setup redirect
prepare redirect msg
sendEncryptedMsg
Party B
party.listen(
options,
connListener
)


{
sessionId: **redirectSessionId**
permanent: **false**,
address: **redirectAddress**
}
handle redirect
add new address to session cache
create connection with redirect host



- Uses the session workflow to establish the connection
- What is different in permanent redirect? The public-key is also sent in the message to the initiator. All future connections will be established with the new host.

http://www.seif.place