

GeneRic Autonomic Signaling Protocol (GRASP) A Tutorial

Brian Carpenter

October 2021

Note

- This tutorial is intended for self-study, not for classroom presentation. Please read it at your own speed.
- There are about 60 slides.
- For general background, before continuing, you are strongly advised to read this article first:

[Autonomic Networking Gets Serious](#), Internet Protocol Journal 24(3), pp2-18, October 2021.

Topics

- Background
- Requirements for an autonomic signaling protocol
- GRASP basics
- GRASP “objectives”
- GRASP operations and messages
- GRASP API
- Logic flows
- Security
- Prototype implementation

Background and General Terminology

- Autonomic Network: Self-managing (self-configuring, self-protecting, self-healing, self-optimizing) with high-level guidance by a central entity (e.g. the Network Operations Center, NOC)
- Autonomic Function: A specific self-managing feature or function.
- Autonomic Service Agent (ASA): An agent that implements an autonomic function, centralized or distributed.
- Autonomic Node: A node that contains at least one ASA and employs autonomic functions
- Autonomic Control Plane (ACP): Self-configuring fully secure virtual network used for all autonomic messaging.

Technical Reading List

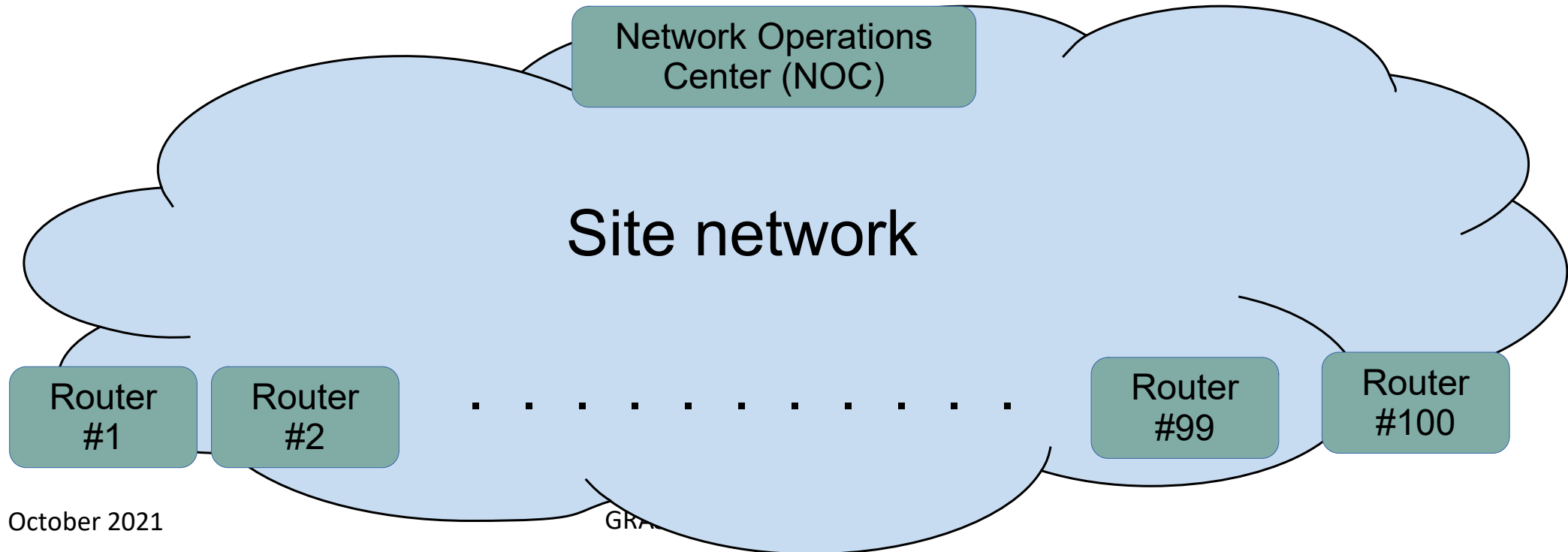
For all technical details, consult the RFCs:

- Background: [RFC7575](#) and [RFC8993](#)
- Specifications
 - Autonomic security: [RFC8994](#) and [RFC8995](#)
 - GRASP itself: [RFC8990](#) (protocol) and [RFC8991](#) (API)
- Use cases:
 - Integration with NOC: [RFC8368](#)
 - IP Prefix management: [RFC8992](#)

Interlude (1)

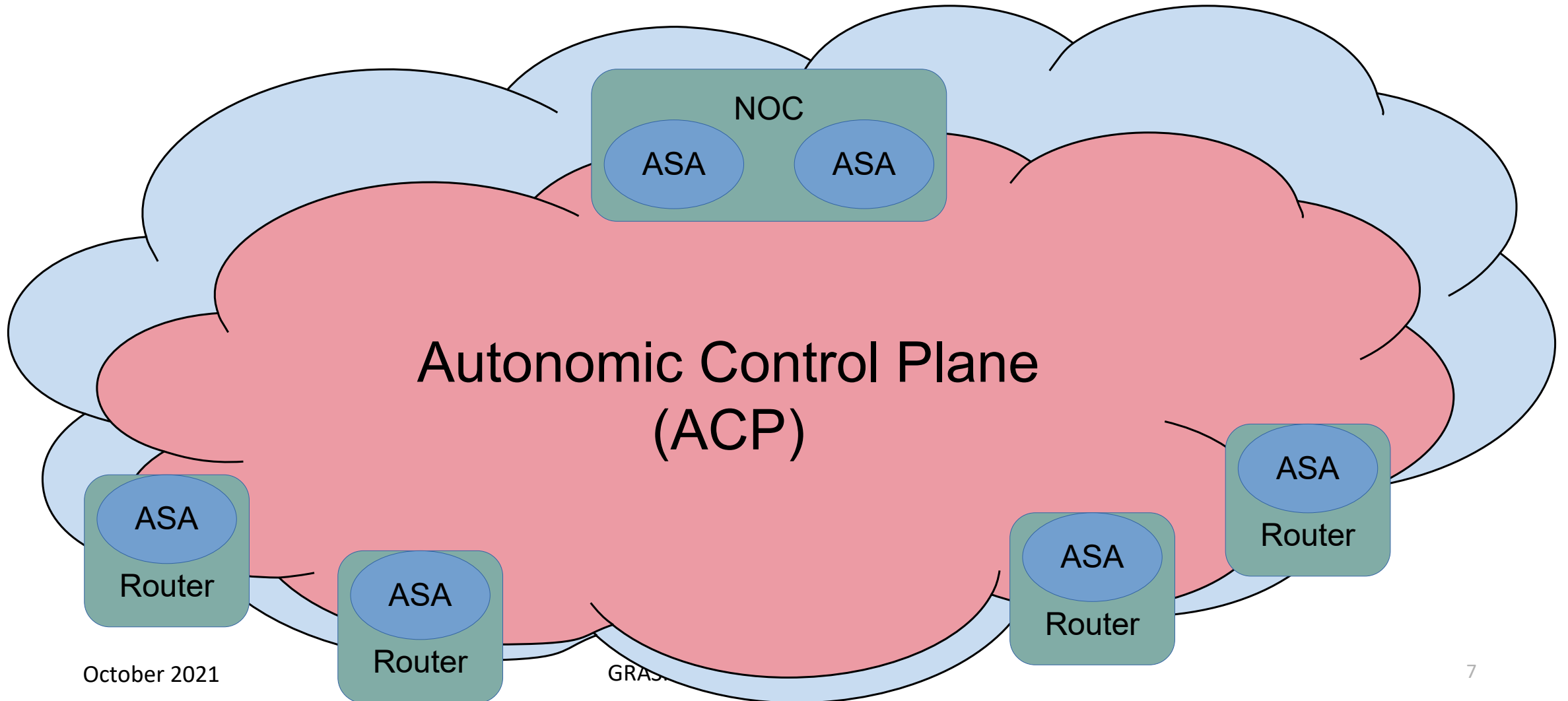


- Imagine a network with a few thousand hosts and (say) a hundred locations. Some resource (e.g., IP address space) must be shared among the locations.
- Can we manage that resource without manual actions?



Interlude (2)

- Add the ACP and some Autonomic Service Agents



Requirements for Autonomic Service Agents (ASAs)

- To act autonomically, ASAs need to communicate with each other.
- Even if policy and resources come from a central origin (the NOC), ASAs need to communicate *peer-to-peer*, especially if the network partitions due to an outage.
- There are two primary forms of communication:
 - Configuration or resource data *sent in one direction* only (from one ASA to others);
 - Configuration or resource data *negotiated* between ASAs.

Why invent a new protocol?

- [RFC8990](#) gives more detail on the ASA communication requirements.
- It also analyzes various existing protocols against these requirements.
- None of them matched up. The IETF ANIMA WG decided that a purpose-designed protocol was the way to go.

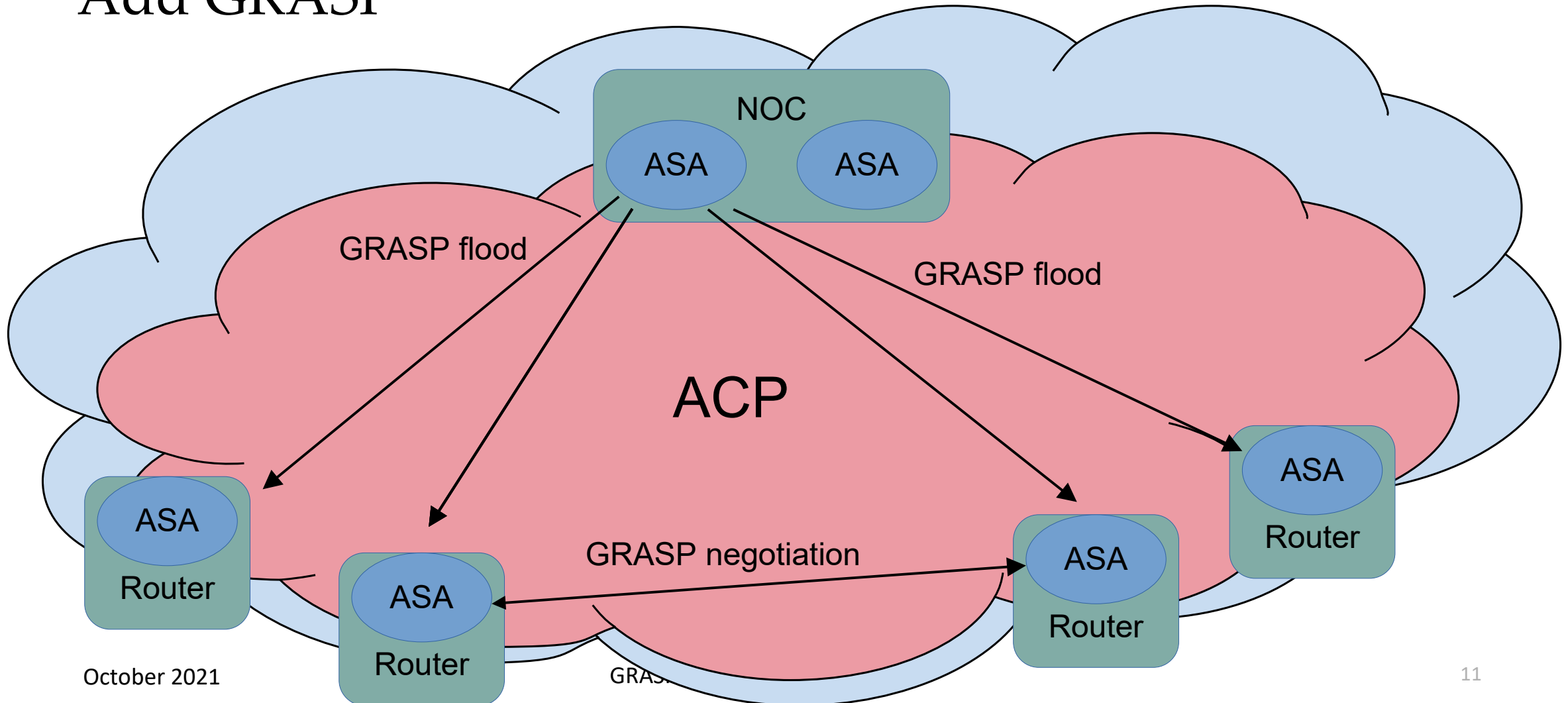
Basics of GeneRic Autonomic Signaling Protocol (GRASP)

- GRASP will be used for signaling between ASAs
 - That includes the special-purpose ASAs that support secure bootstrap & Autonomic Control Plane (ACP) creation
 - After that, GRASP runs over the ACP to guarantee security
- GRASP provides discovery, flooding, synchronization and negotiation for the technical “objectives” supported by ASAs
 - Based on CBOR (Concise Binary Object Representation)
 - Objectives can be expressed in JSON or Python-like syntax & semantics

Interlude (3)



- Add GRASP



Why CBOR? Why not TLV?

- The earliest design used a traditional type-length-value format.
- However, switching to CBOR¹ ([RFC8949](#)) gave
 - much greater flexibility and extensibility,
 - no performance loss,
 - clear protocol definitions in CDDL² ([RFC8610](#)),
 - easy mapping to JSON when useful,
 - easy mapping to modern languages like rust and Python.

¹Concise Binary Object Representation

²Concise Data Definition Language

GRASP Technical Objectives

- In GRASP, an *objective* is a configurable parameter:
 - a logical, numerical or string value, or a more complex data structure.
 - used in Discovery, Negotiation, Flooding and Synchronization.
 - semantics depend on the autonomic function concerned, and are built into the code of each ASA.
- Example for IP prefix management:

```
["PrefixManager", flags, loop_count,  
                             [IP_version, prefix_length, prefix]]
```

Formal syntax of a GRASP Objective (1)

**objective = [objective-name, objective-flags,
loop-count, ?objective-value]**

objective-name = text

objective-value = any

loop-count = 0..255

Formal syntax of a GRASP Objective (2)

objective-name = text

This is any human-readable UTF-8 string.

- Generic objectives **MUST NOT** include a colon (":") and **MUST** be registered with IANA.
- Privately defined objectives **MUST** include at least one colon (":"). The string preceding the last colon in the name **MUST** be globally unique, such as a fully-qualified DNS name.

Formal syntax of a GRASP Objective (3)

objective-flags

A byte containing up to 8 flag bits. Bit numbers defined so far:

F_DISC: 0 ; valid for discovery

F_NEG: 1 ; valid for negotiation

F_SYNCH: 2 ; valid for synchronization

F_NEG_DRY: 3 ; negotiation is a dry run

F_NEG and F_SYNCH cannot both be set to 1.

Formal syntax of a GRASP Objective (4)

objective-value = any

GRASP does not restrict the value field of an objective. Anything that can be expressed in CBOR can be used: for example, a single integer, a UTF-8 string, an array of floating point numbers, or any kind of JSON-like object.

In other words, whatever suits the configuration or optimization task of an ASA is OK.

The specification of a given GRASP objective must define the format of the value field, preferably using CDDL for clarity.

Formal syntax of a GRASP Objective (5)

loop-count = 0..255

In a discovery operation, this variable is used to limit the scope of discovery (see later).

In a negotiable objective ($F_NEG = 1$), this variable counts down at each step of a negotiation, and the negotiation fails if it reaches zero.

In a synchronizable objective ($F_SYNCH = 1$), this variable is used to limit the scope of a flooding operation (see later).

A brown silhouette of a coffee cup with three wavy lines above it representing steam. The cup sits on a small rectangular base.

- The flag byte:
F_DISC and
F_NEG set

The loop count

Prefix length

The actual binary prefix

GRASP Operations and Messages

- **Discovery** is used by any ASA that needs to discover another (peer) ASA that supports a given objective. It returns zero, one or more responses.
- **Negotiation** is used between two ASAs that support a given objective. They swap values of the objective until negotiation succeeds or fails.
 - GRASP does not support multiparty negotiation; negotiation is 1-to-1.
- **Synchronization** is used between any pair of ASAs that support a given objective. One of them obtains a value of the objective from the other.
- **Flood Synchronization** is used when one ASA needs to distribute the value of a given objective to all others.
 - GRASP does not currently support selective distribution.

The next slide lists the message types that support these operations.

GRASP Message Types

Discovery (multicast)	M_DISCOVERY
Discovery Response	M_RESPONSE
Request Negotiation	M_REQ_NEG
Negotiation	M_NEGOTIATE
Confirm Waiting	M_WAIT
Negotiation End	M_END
Request Synchronization	M_REQ_SYN
Synchronization	M_SYNCH
Flood Synchronization (multicast)	M_FLOOD
Invalid	M_INVALID
No operation	M_NOOP

Transport and IP Layer Usage

- Multicasts are IPv6 link-local to port 7017.
 - All modern operating systems support link-local IPv6 by default; nothing to configure.
 - When necessary, GRASP nodes relay these multicasts on other links.
- Unicast messages use a reliable transport protocol over IPv6.
 - Depending on the security environment provided by the ACP, this may be TLS.
 - Otherwise, TCP.
- In a deployment with the standard ACP, the IPv6 environment is self-creating, using Unique Local Addresses, with no operator intervention.

Formal syntax of a GRASP message (1)

**message-structure = [MESSAGE_TYPE, session-id,
 ?initiator,
 *grasp-option]**

MESSAGE_TYPE = 0..255

session-id = 0..4294967295 ; up to 32 bits

grasp-option = any

Formal syntax of a GRASP message (2)

MESSAGE_TYPE = 0..255

Just an integer defining the message type.

session-id = 0..4294967295 ; up to 32 bits

A unique pseudo-random number identifying a session (discovery, negotiation, etc.). Together with the IP address of the **initiator** of a session, this forms a unique handle, to distinguish simultaneous sessions.

Formal syntax of a GRASP message (3)

grasp-option = any

Defined GRASP options *include*:

objective (as above)

**ipv6-locator-option = [O_IPv6_LOCATOR, ipv6-address,
transport-proto, port-number]**

(used in M_RESPONSE and elsewhere)

accept-option = [O_ACCEPT]

decline-option = [O_DECLINE, ?reason]

(used in M_END to finish a negotiation)

Interlude (5)



- M_NEGOTIATE message in detail

Message type

Session
identifier

```
[5, 1763418432,  
 ['PrefixManager', 3, 6,  
  [6, 57, b'20010db8ffffffff8000000000000000000000']]]
```

GRASP objective
as before

Interlude (6)

- CBOR as it appears on the wire

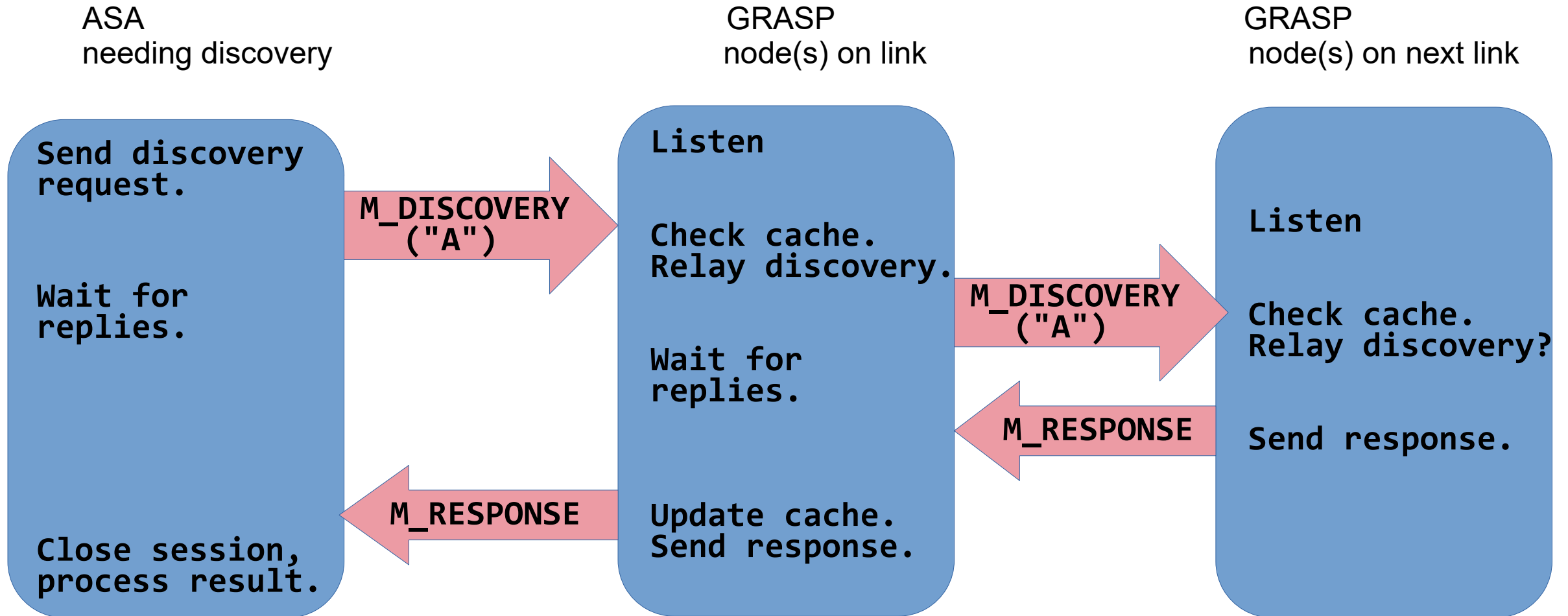
```
b'4df6aa69862ca8d0fc0b2dcd88c726888bf905b7a2fb3a  
8ba501606e35932dadb7474a3ef5c43c3a9a5e58fbcbbc95  
1d'
```

- 48 bytes

What happens during Discovery

- An ASA that needs to find a peer handling objective “A” originates an M_DISCOVER message for objective “A”.
 - This goes out as a link-local UDP multicast on each of the node’s interfaces to the ACP.
- Every GRASP node that receives the multicast and supports “A” or has cached the address of a node that supports “A” sends a unicast M_RESPONSE back, including an **ipv6-locator-option**.
- Any node which is also a router to other links will relay the M_DISCOVER to its other interfaces, and then relay back any M_RESPONSEs that it receives (and cache them).
- The original node will return all results received before a specified timeout to the requesting ASA (and cache them).

A discovery process

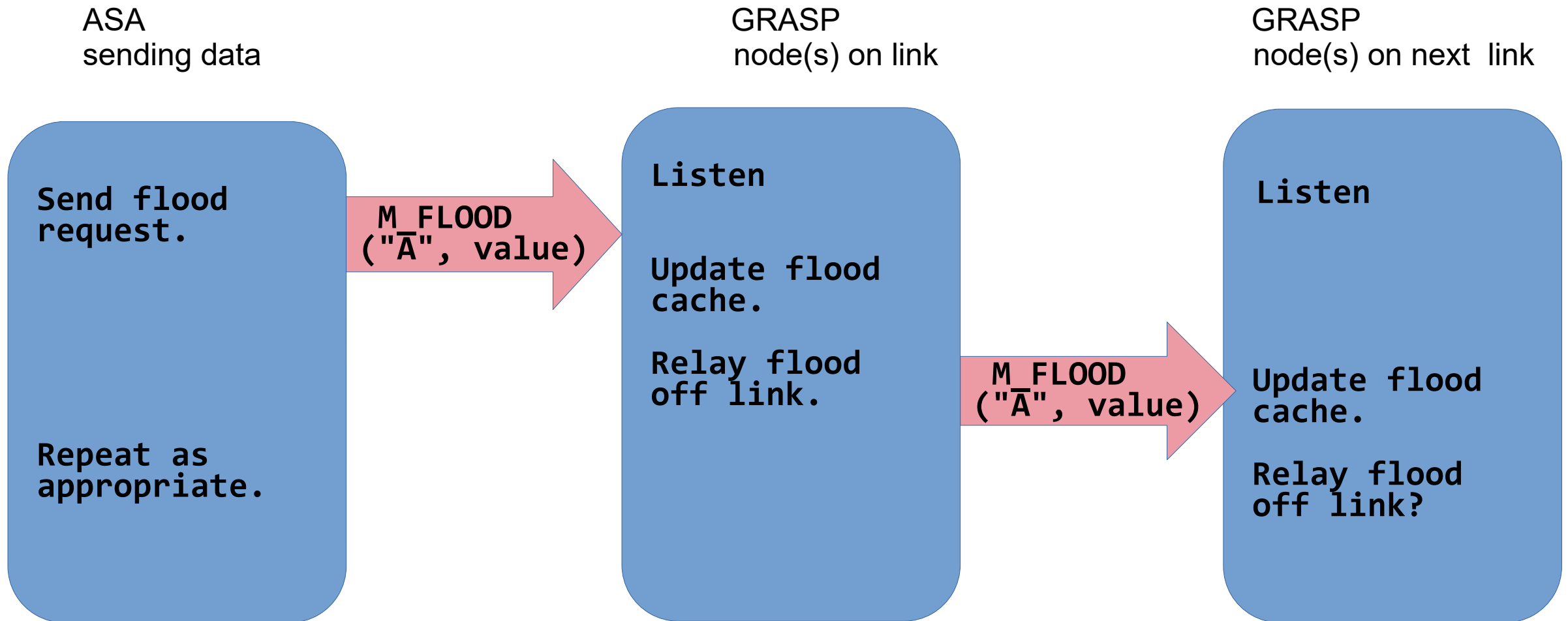


(Relaying is limited by loop count.)

What happens during Flooding

- An ASA handling objective “A” originates an M_FLOOD message for objective “A”, including whatever value of the objective it wants to send to all nodes. This goes out as a link-local UDP multicast on each of the node’s interfaces to the ACP.
- Every GRASP node that receives the multicast caches a copy of the objective and its value for local use.
- Any node which is also a router to a different physical link will also relay the M_FLOOD to its other interfaces.
- Floods may specify an expiry timeout, after which other nodes will mark the cached value as expired.

A flooding process



(Relaying is limited by loop count.)

What happens during Synchronization

- An ASA able to provide a value for objective “A” waits for any incoming M_REQ_SYN message. This also makes the ASA discoverable for “A”.
 - This is server-like behavior
- An ASA needing to obtain a value for “A” first uses discovery to find all peers that support “A” and chooses one of them. It then originates a unicast M_REQ_SYN to the chosen peer.
- The peer responds with a unicast M_SYNCHRONIZE message containing the current value of “A”.

(A diagram for synchronization comes later.)

What happens during Negotiation

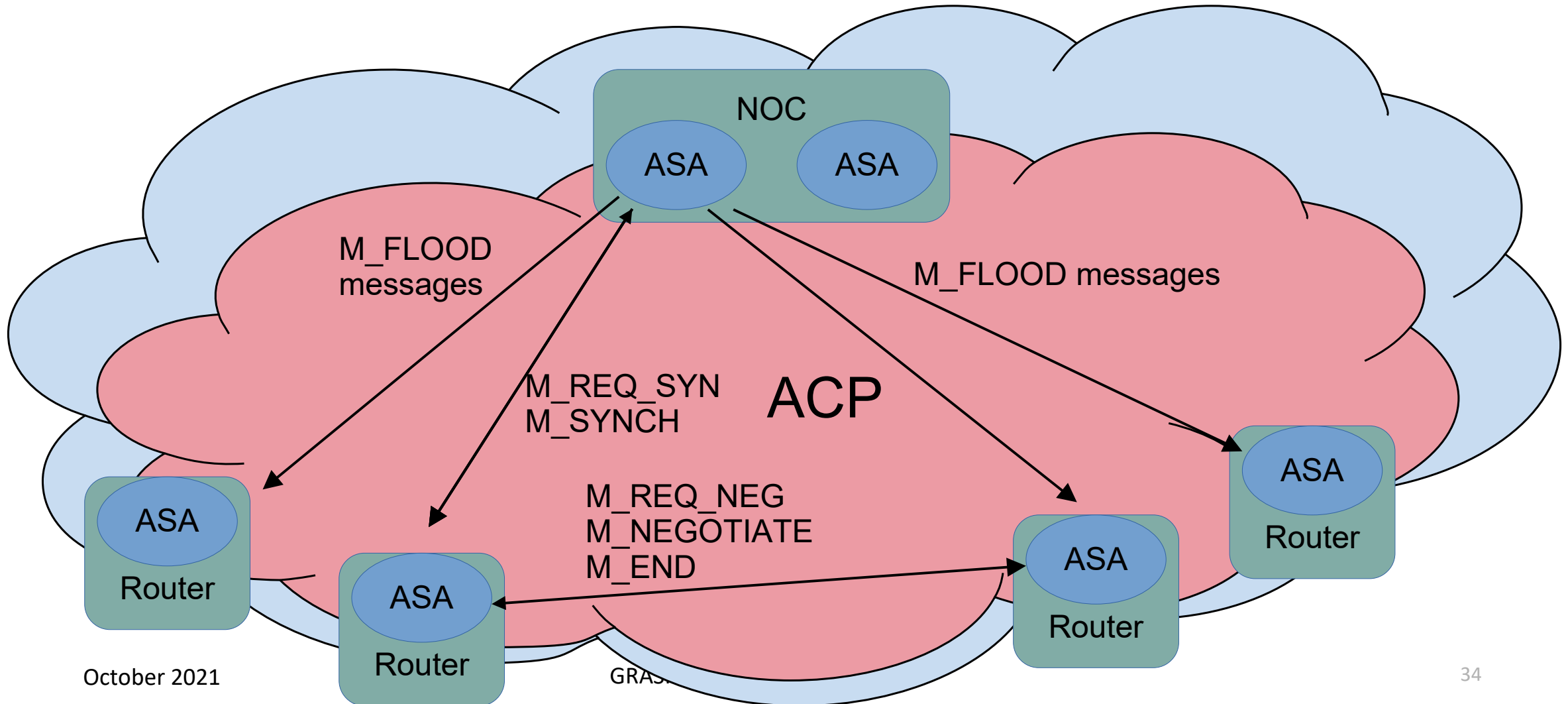
- An ASA able to negotiate a value for objective “A” waits for any incoming M_REQ_NEG message. This also makes the ASA discoverable for “A”.
 - This is server-like behavior
- An ASA needing to negotiate a value for “A” first uses discovery to find all peers that support “A” and chooses one of them. It then originates a unicast M_NEGOTIATE to the chosen peer.
- The two peers swap alternate M_NEGOTIATE messages containing a proffered value of “A” until one of them ends the negotiation with M_END (carrying O_ACCEPT or O_DECLINE).
 - Note that a failed negotiation is not a protocol error.
- Either peer can insert an M_WAIT message to delay the timeout.

(A diagram for negotiation comes later.)

Interlude (7)



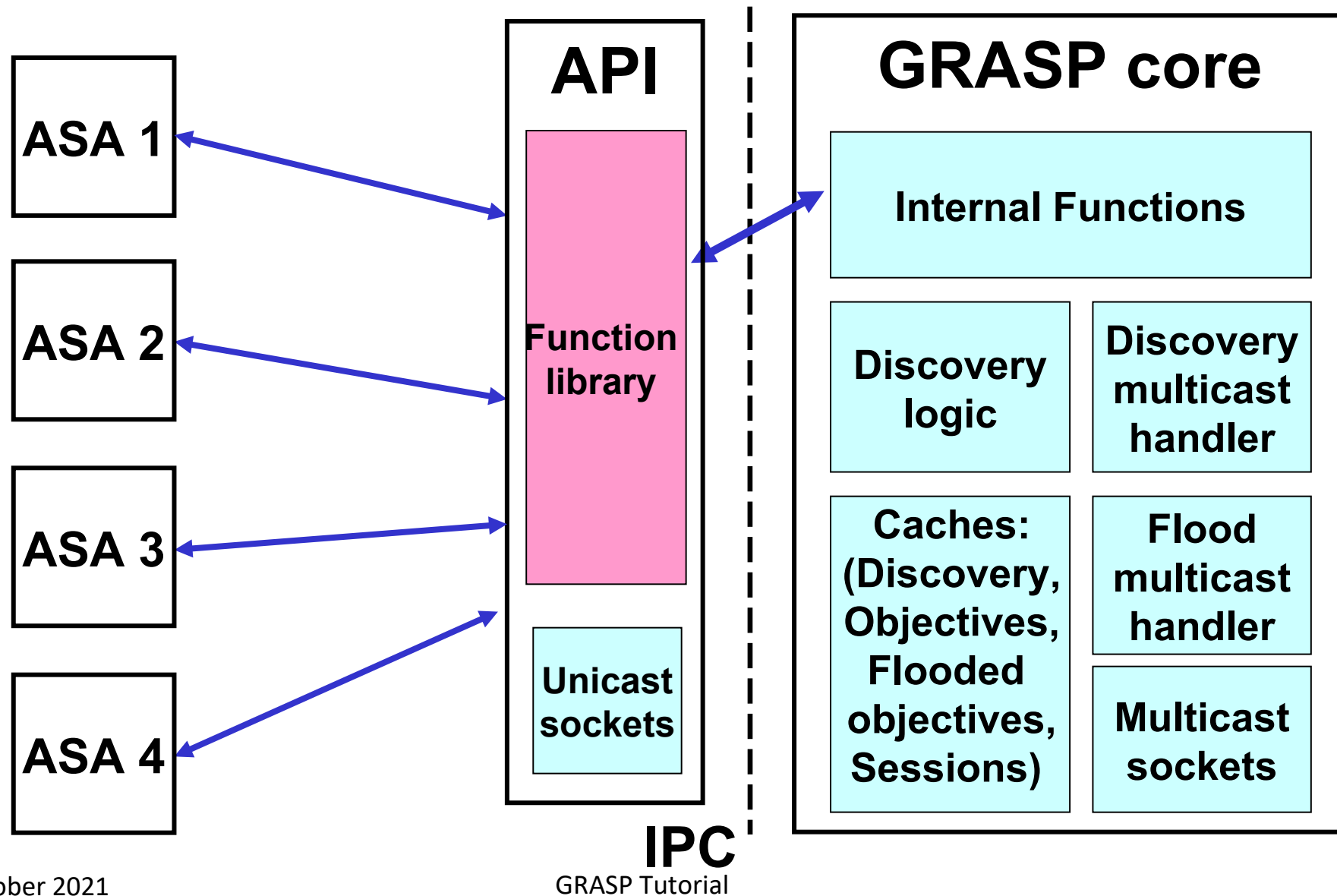
- Add objectives and messages



Confused? Then you need an API.

- Although GRASP has few message types, it has quite powerful capabilities:
 - Discovery
 - Flooding
 - Client/server data synchronization
 - Peer-to-peer data negotiation
- Each of these has a clear purpose in autonomic functions.
- Each has its own complexity.
- In some cases, a subset of GRASP will be built into an application. Often, it will appear as an API. We will now discuss the ASA programmer's view of this API.

ASA programmer's view



GRASP API Functions

- *Registration*. An ASA can register itself and register the GRASP Objectives it manipulates.
- *Discovery*. An ASA can discover a peer willing to respond for a particular objective.
- *Negotiation*. An ASA can act as an initiator (requester) or responder (listener) for a negotiation session. Negotiation is a symmetric process, so most functions can be used by either party.
- *Synchronization*. An ASA can act as an initiator (requester) or responder (listener and data source) for data synchronization.
- *Flooding*. An ASA can send and receive a GRASP Objective that is flooded to all nodes of the ACP.

The following slides show a simplified Python rendering of the main API functions.

Data types

class objective(name)

Attributes include **.loop_count** and **.value**

class asa_locator()

Attributes include **.locator** and **.is_ipaddress**

class tagged_objective()

Attributes **.objective** and **.locator**

Registration Functions

register_asa(asa_name)

Tells GRASP that a new ASA is starting.

Returns **(zero, asa_handle)** if successful,
(errorcode, None) if failure.

The ASA must use **asa_handle** in every subsequent call.

register_obj(asa_handle, objective)

Tells GRASP that the ASA will support the given objective.

Returns **(zero)** if successful,
(errorcode) if failure.

Discovery Function

discover(asa_handle, objective, timeout)

Returns **(zero, list of asa_locator)** if successful,
(errorcode, []) if failure.

Flooding Functions

flood(asa_handle, ttl, tagged_objectives)

Floods a list of objectives with values to all nodes in the autonomic network.

Returns **zero** if successful,
errorcode if failure.

get_flood(asa_handle, objective)

Fetches flooded objectives from local cache.

Returns **(zero, tagged_objectives)** if successful,
(errorcode, None) if failure.

Synchronization Functions

listen_synchronize(*asa_handle*, *objective*)

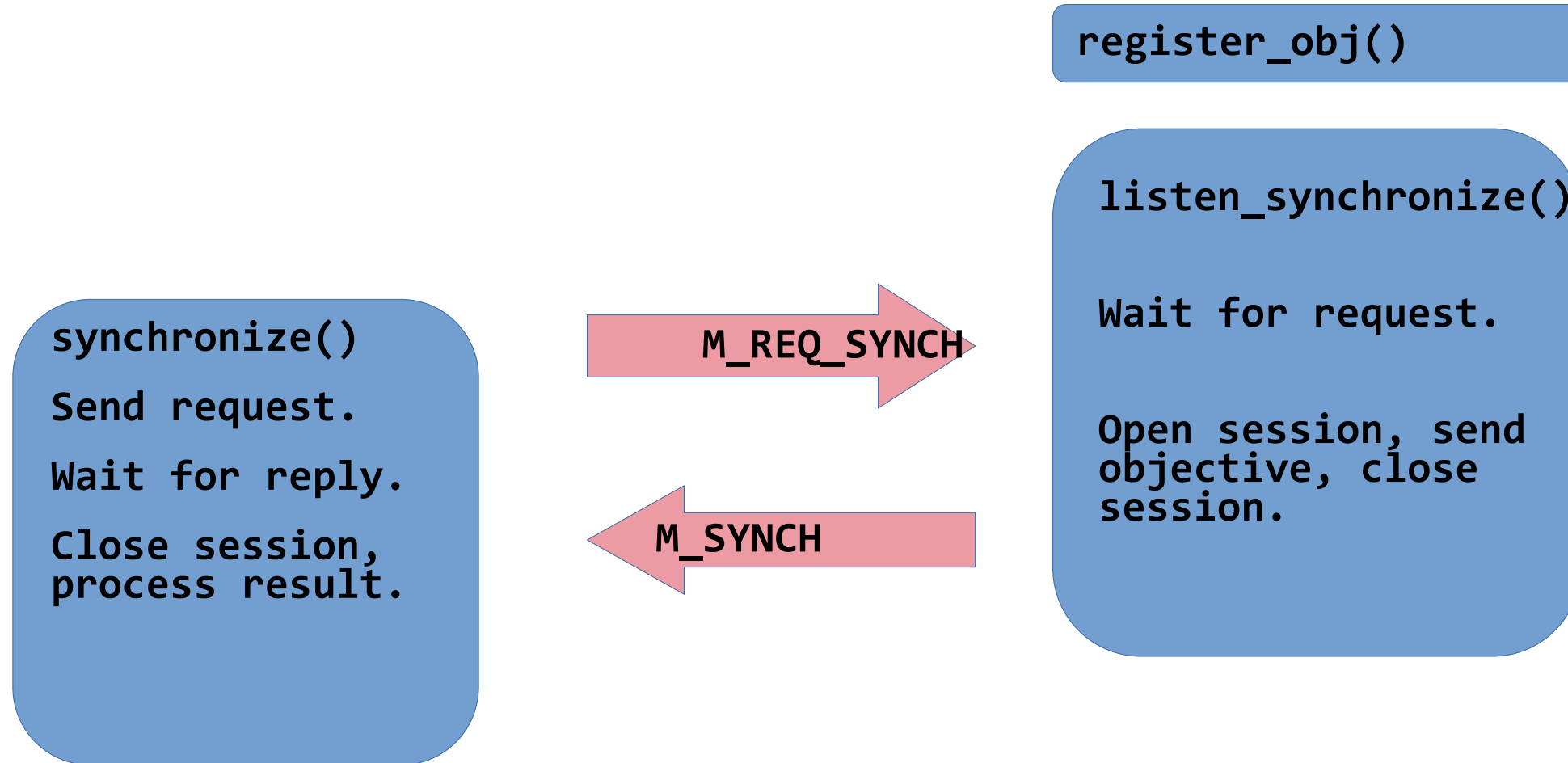
GRASP will listen for incoming synchronization requests and reply with the given objective and its value. Must be a separate thread.

synchronize(*asa_handle*, *objective*, *locator*, *timeout*)

Requests synchronized value of the given objective. The **locator** is an **asa_locator** as returned by **discover()**.

Returns **(zero, objective)** if successful,
(errorcode, None) if failure.

A synchronization session



Negotiation Functions (1)

listen_negotiate(asa_handle, objective)

Listen for incoming requests and start a negotiation session. Must be a separate thread.

Returns (**zero**, **session_handle**, **requested_objective**) if successful.

The **session_handle** must be used in subsequent calls.

The value of the **requested_objective** is the peer's initial offer for negotiation.

Negotiation Functions (2)

request_negotiate(*asa_handle*, *objective*, *peer*, *timeout*)

Requests negotiation of the given objective, starting with its current value. The **peer** is an **asa_locator** as returned by **discover()**.

There are 4 possible returns:

(zero, None, objective, None) The peer agreed with the offered value.

(zero, session_handle, objective, None)

Negotiation continues. The returned objective contains the value offered by the peer.

(errors.declined, None, None, string)

The peer declined further negotiation, the string gives a reason

(errorcode, None, None, None) Some other error.

Negotiation Functions (3)

After **request_negotiate()** various functions are used symmetrically by either side:

**negotiate_step(asa_handle, session_handle,
 objective, timeout)**

Sends the next proffered value of the objective to the peer. The returns are identical to **request_negotiate()**. Used alternately by the two peers.

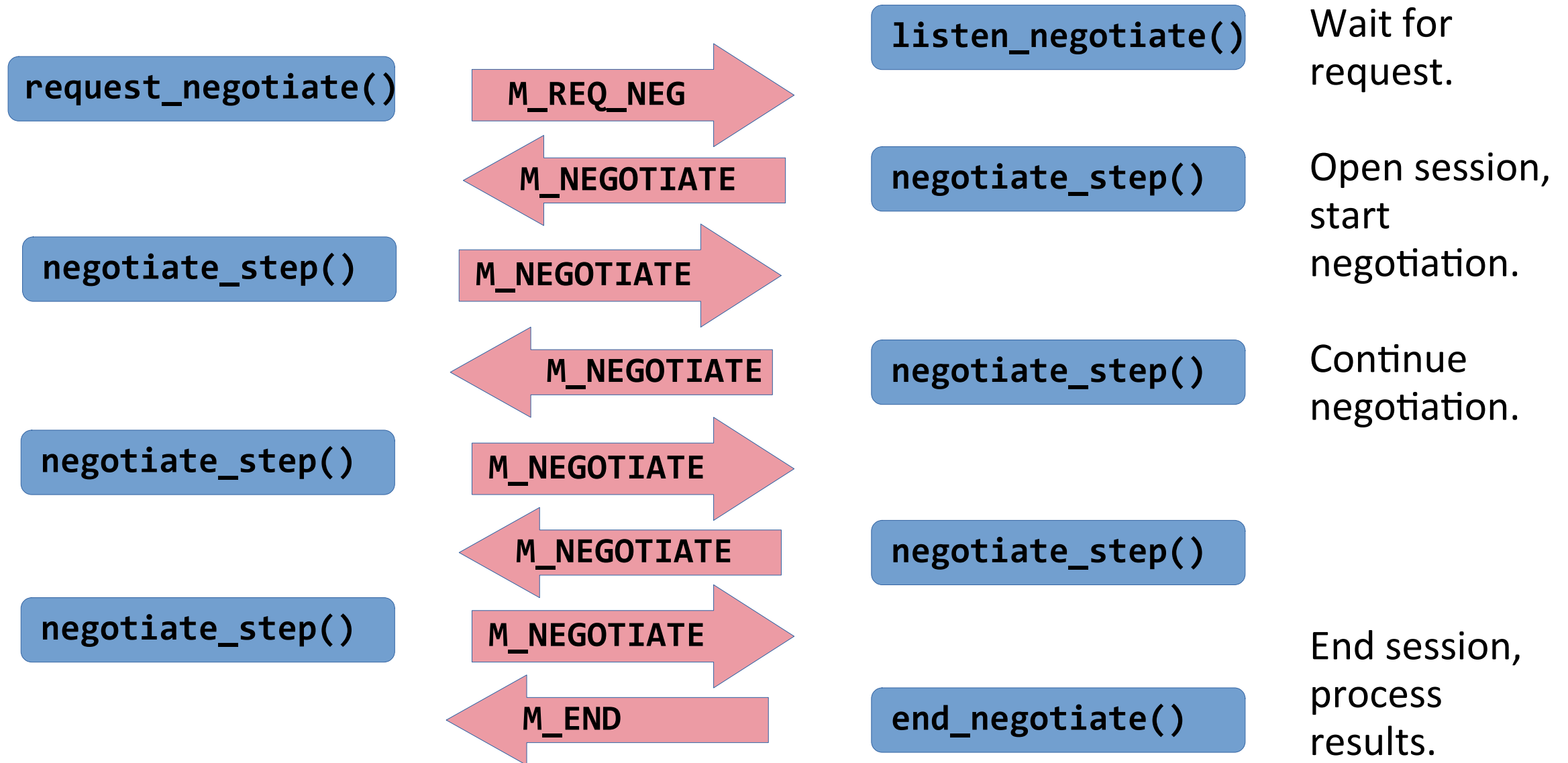
negotiate_wait(asa_handle, session_handle, timeout)

Extend the timeout.

end_negotiate(asa_handle, session_handle, result)

End negotiation (**result=True** for success, **False** to decline further negotiation.)

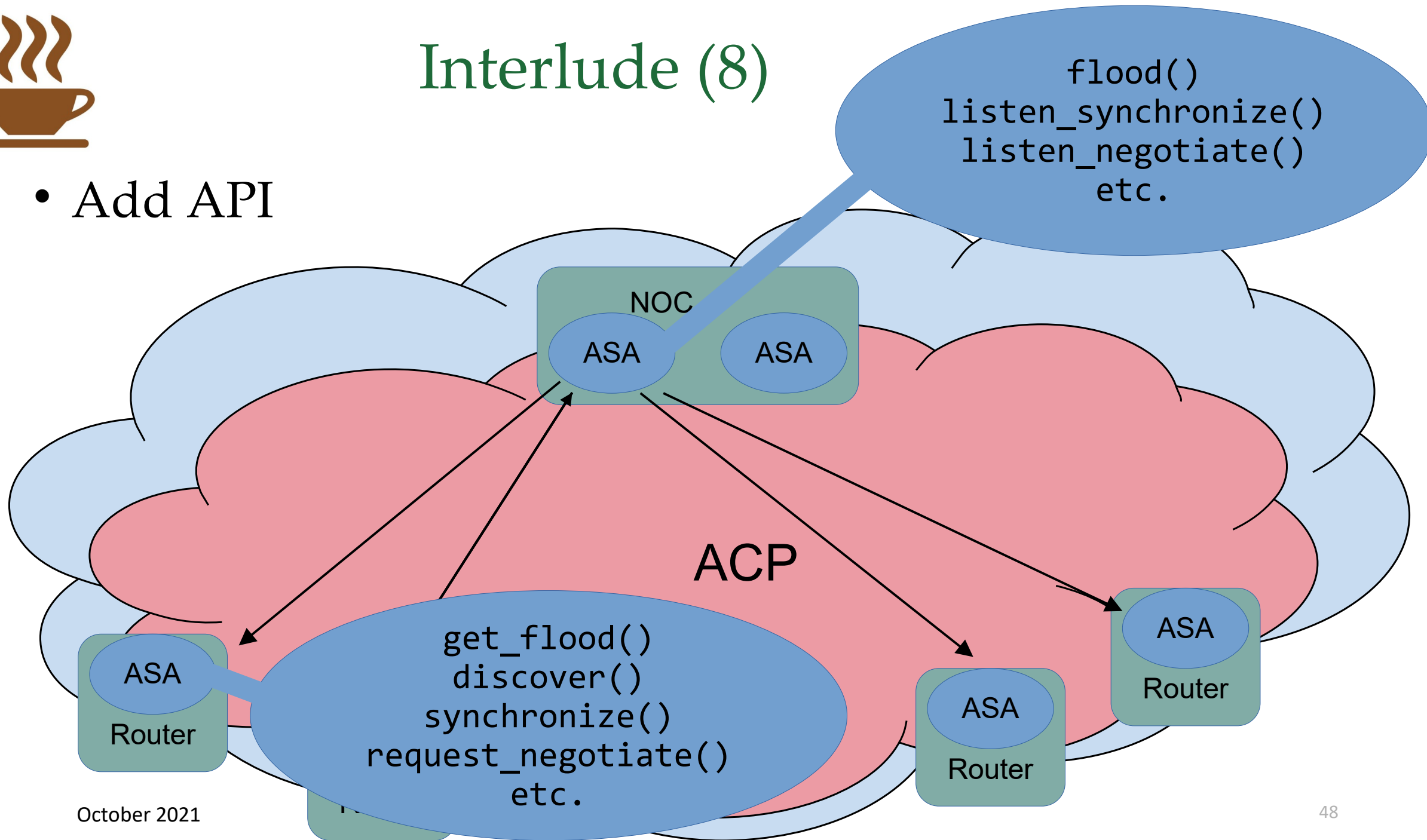
A negotiation session





Interlude (8)

- Add API



Logic flows

- The API is designed for use in asynchronous operations such as handling multiple simultaneous negotiations, or processing floods, synchronizations, and negotiations at the same time.
- This could be done using an event-loop mechanism or a threading mechanism, depending on the programming environment in use.
- In the following slides, we show logic flows for an ASA that manages some (unnamed) distributed resource
 - We assume a threaded model
 - A very general outline is followed by pseudocode

Logic flow outline – main thread

```
MAIN thread:
initialise resource pool
if origin:
    start FLOODER to broadcast parameters
start NEGOTIATOR and GARBAGE_COLLECTOR
if not origin:
    get resource parameters flooded by GRASP
    start ASSIGN thread (allocates resources)
do forever:
    if resource pool is low:
        negotiate for more resource from GRASP peer(s)
```

Logic flow outline – other threads

FLOODER thread:

periodically flood resource parameters to all GRASP nodes

NEGOTIATOR thread:

wait for and satisfy negotiation requests from GRASP peers

GARBAGE_COLLECTOR thread:

periodically compact the resource pool

DELEGATOR thread:

manage resource requests from non-autonomic devices & applications, assign resources from pool

Pseudocode: MAIN (1)

Initialization

Create empty resource_pool

register_asa()

register_obj("EX1.Resource")

register_obj("EX1.Params")

if origin:

 Obtain initial resource_pool contents from NOC

 Obtain value of EX1.Params from NOC

 Start FLOODER thread to flood EX1.Params

 Start SYNCHRONIZER listener for EX1.Params

Start MAIN_NEGOTIATOR thread for EX1.Resource

if not origin:

 get_flood("EX1.Params")

 Start DELEGATOR thread

Start GARBAGE_COLLECTOR thread

Pseudocode: MAIN (2)

```
# main loop
do forever:
    if resource_pool is low:
        peers = discover("EX1.Resource")
        peer = #any choice among peers
        request_negotiate("EX1.Resource", peer)
        #Wait for response (M_NEGOTIATE, M_END or M_WAIT)
        if OK:
            if offered amount of resource sufficient:
                end_negotiate(True)
                Add resource to pool
            else:
                end_negotiate(False) #negotiation failed
        sleep() #sleep time depends on application scenario
```

This is a very simple use of negotiation because it doesn't loop.

(A full negotiation needs an outer loop here.)

Pseudocode: NEGOTIATOR

```
# MAIN_NEGOTIATOR thread:
do forever:
    listen_negotiate("EX1.Resource") # wait for M_REQ_NEG
    Start a separate new NEGOTIATOR thread for requested amount A

# NEGOTIATOR thread:
Request resource amount A from resource_pool
if not OK:
    while not OK and A > Amin:
        A = A-1
        Request resource amount A from resource_pool
if OK:
    negotiate_step("EX1.Resource") # Offer resource amount A
    if received M_END + O_ACCEPT:
        # negotiation succeeded
    elif received M_END + O_DECLINE or other error:
        # negotiation failed
        Return resource amount A to resource_pool
else:
    end_negotiate(False) # negotiation failed
```

Will offer the best
it can get from
the resource
pool.

Again, a single
step
negotiation

Pseudocode: DELEGATOR

```
# There are no GRASP calls. This is actual resource assignment.
do forever:
    Wait for request or release for resource amount A
    if request:
        Get resource amount A from resource_pool
        if OK:
            Delegate resource to consumer
            Record in delegated_list
        else:
            Signal failure to consumer
            Signal main thread that resource_pool is low
    else:
        Delete resource from delegated_list
        Return resource amount A to resource_pool
```

Pseudocode: other threads

SYNCHRONIZER thread:

do forever:

listen_synchronize("EX1.Params")

FLOODER thread:

do forever:

flood("EX1.Params")

sleep() #sleep time depends on application scenario

GARBAGE_COLLECTOR thread:

do forever:

Search resource_pool for adjacent resources

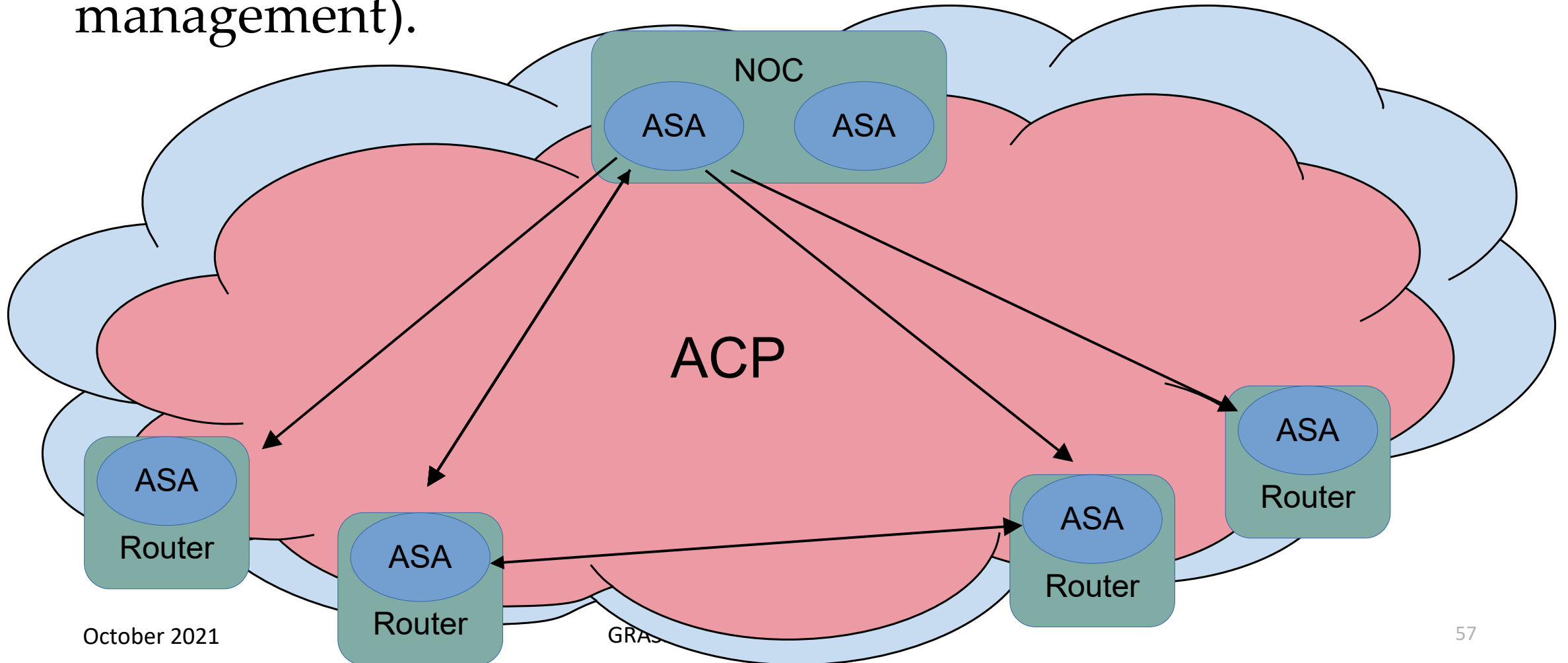
Merge adjacent resources

sleep() #sleep time depends on application scenario

Final Interlude



- With those logic flows, we have designed the ASAs for a distributed resource manager (such as IP prefix management).



Security

- GRASP does not have its own security mechanism. It is used over a secure and encrypted Autonomic Control Plane.
- TLS is recommended for the unicast messages.
- GRASP includes a very restricted subset for use during bootstrapping of the ACP, known as “Discovery Unsolicited Link-Local” (DULL).

GRASP Prototype

- A Python 3 implementation of GRASP and its API as **graspi.py**
- About 2600 lines of code
- A test suite to exercise as many code paths as possible
- Various toy ASAs to test "real" operation across the network
 - bank/client negotiation
 - model of secure bootstrap process
 - model of IPv6 prefix management
 - bulk transfer using GRASP

<https://github.com/becarpenter/graspy>

Start with

<https://github.com/becarpenter/graspy/blob/master/graspy.pdf>

The End

- Raise issues, questions and comments at <https://github.com/becarpenter/grasdoc/issues>
- Join the ANIMA WG at <https://www.ietf.org/mailman/listinfo/anima>
- Want to improve the Python prototype? It's open source on GitHub at <https://github.com/becarpenter/graspy>
- Got your own GRASP implementation? List it at <https://brski.org/grasp-impls.html>