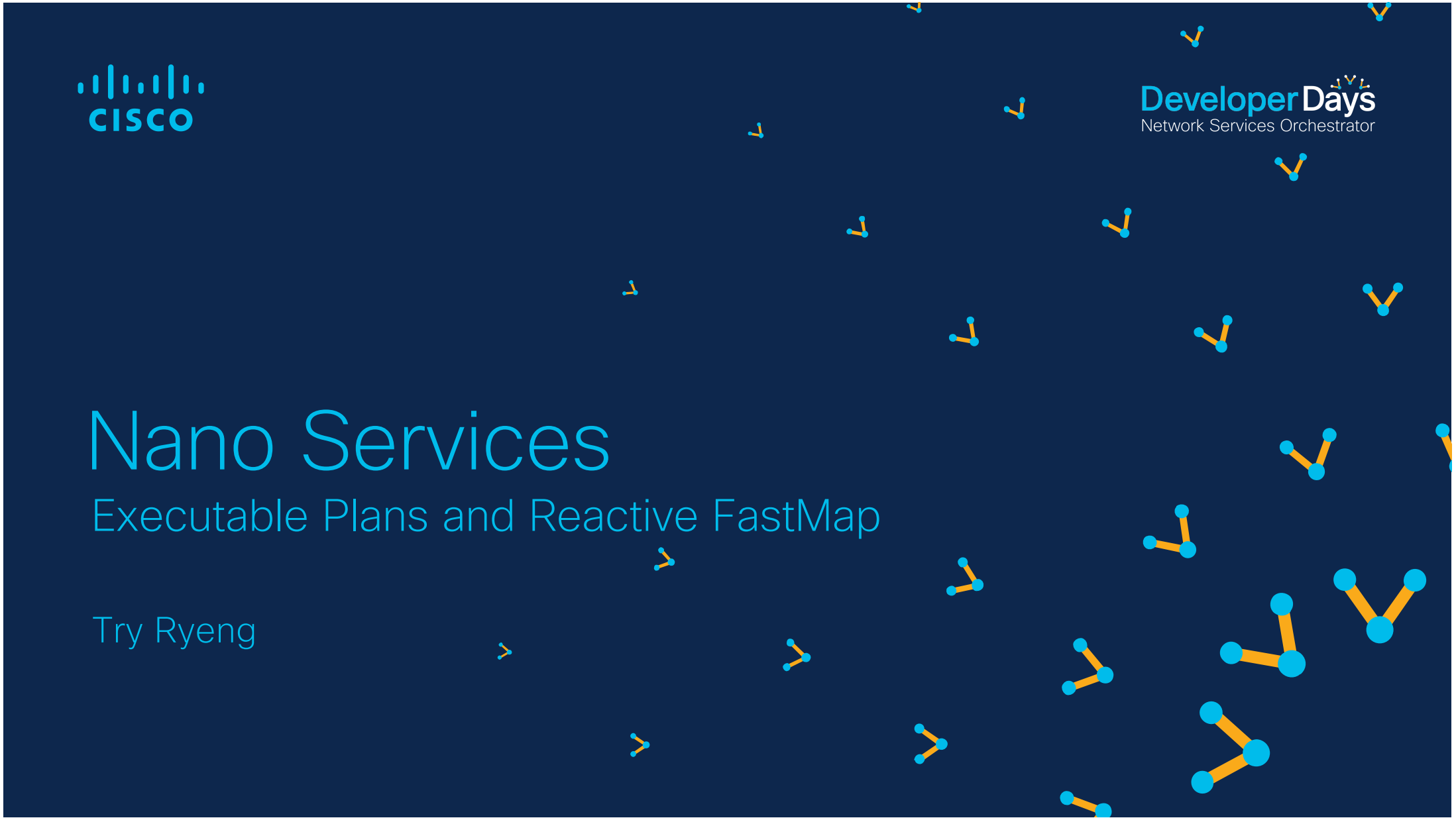

cisco
DeveloperDays
Network Services Orchestrator
Nano Services
Executable Plans and Reactive FastMap
Try Ryeng

What is FastMap?

1. The ability to monitor and store a service create config data.

2. Handle service delete by removing the stored create config data.

3. Handle service update as a delete + create followed by calculating the actual changes from latest “create”.

What is a service re-deploy?

Ability to recalculate a service like an update even though no service parameters are changed.

For a normal service this is expected to be a no-op.

However, if the service code produces new config these changes will be set.

What is a Reactive FastMap (RFM) service?

Reactive FastMap is a design pattern.

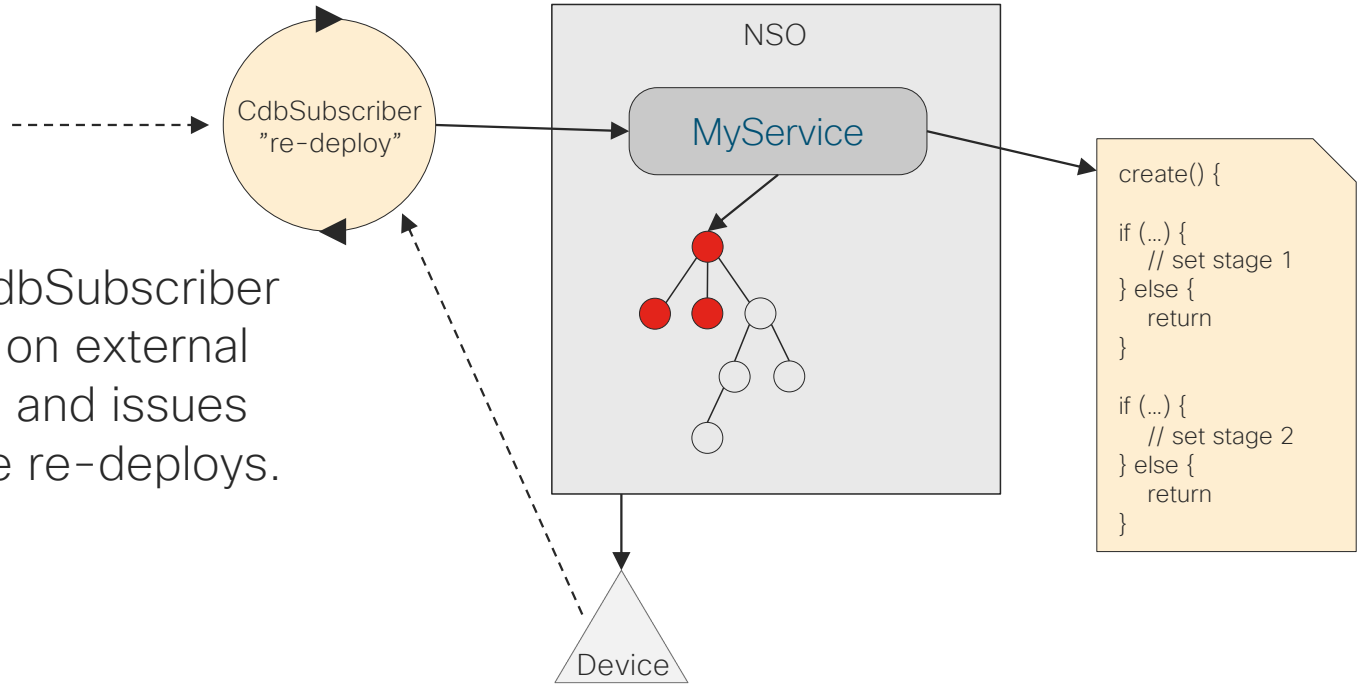
For service scenarios that include asynchronous behavior, actions, rpc-calls etc. that are not allowed in the FastMap algoritm.

RFM depends heavily on re-deploys.

Anatomy of a RFM service

- The Service create() is structured in conditional stages. Normally some service specific operational data governs which stages of the service code can be executed at any specific time.
- Some outside mechanism is added that can re-deploy the service when some event/stimuli occurs. Typically this is implemented as a CDB subscriber. This is also the place where all asynchronous stuff goes.

Anatomy of a RFM service in pictures.



The CdbSubscriber reacts on external events and issues service re-deploys.

The create() code is structured so that it only sets the device config that is allowed up to the current stage

What is a RFM service plan?

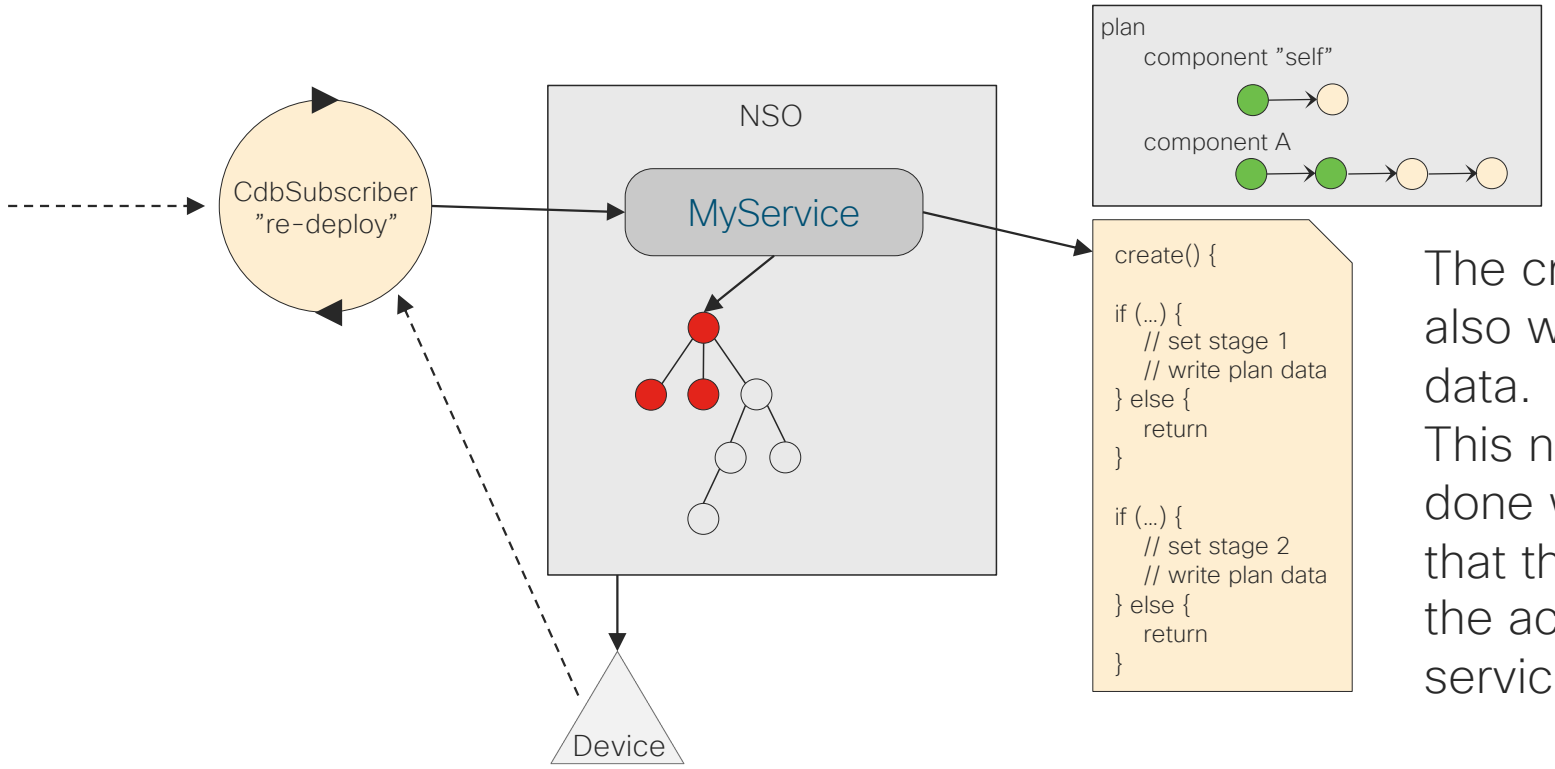
Structured operational data that shows the progress of a RFM service.

Answers the question of when a RFM service is fully converged.

RFM service plan data

- Consist of one or many plan components which in turn consists of one or many plan states.
- A plan component must have “ncs:init” as its first state and “ncs:ready” as its last. It can have any arbitrary number of states in between.
- A state have a status with a predefined set of values (not-reached, reached or failed).
- A service must have an “ncs:self” component that shows the overall progress.

Traditional RFM service with plan data



The create() code also writes the plan data.
This needs to be done with care so that the plan reflects the actual state of the service.

Traditional RFM code example

At least a CDB subscriber and a service create implementation is needed.

A Simple Traditional RFM Service

```
public class AllocVlanServiceRFS {

    @ServiceCallback(servicePoint = "alloc-vlanspnt",
        callType = ServiceCBType.CREATE)
    public Properties create(ServiceContext context,
                             NavuNode service,
                             NavuNode root,
                             Properties opaque) throws DpCallbackException {

        try {
            NavuList managedDevices = root.
                container("devices").list("device");
            for (NavuContainer device : managedDevices) {
                if (device.list("capability").isEmpty()) {
                    String mess = "Device %1$s has no known capabilities, " +
                                  "has sync-from been performed?";
                    String key = device.getKey().elementAt(0).toString();
                    throw new DpCallbackException(String.format(mess, key));
                }
            }
        } catch (DpCallbackException e) {
            throw e;
        } catch (Exception e) {
            throw new DpCallbackException("Not able to check devices", e);
        }

        NavuContext operCtx = null;
        try {
            System.out.println("CREATE()");
            operCtx = new NavuContext(root.context().getMaapi());
            int th = operCtx.startOperationalTrans(Conf.MODE_READ);
            NavuContainer operBase = new NavuContainer(operCtx);
            NavuContainer operRoot = operBase.container(allocVlanService.hash);

            NavuNode vlan = service;
            ConfValue serviceKey = vlan.leaf("name").value();

            NavuList dataList = vlan.getParent().getParent().
                list("alloc-vlan-data");
            NavuContainer data = dataList.sharedCreate(serviceKey);


            NavuContainer getUnit = data.container("request-allocate-unit").
                sharedCreate();

            ConfValue unitVal = null;
            try {
                unitVal = operRoot.list("alloc-vlan-data").
                    elem(serviceKey.toString()).
                    container("request-allocate-unit").
                    leaf("unit").value();
            }
            catch (Exception e) {}
            System.out.println("unitVal = " + unitVal);

            ConfValue vlanVal = null;
```

```
            if (unitVal != null) {
                NavuContainer getVid = data.container("request-allocate-vid").
                    sharedCreate();

                try {
                    vlanVal = operRoot.list("alloc-vlan-data").
                        elem(serviceKey.toString()).
                        container("request-allocate-vid").
                        leaf("vlan-id").value();
                }
                catch (Exception e) {}
            }

            System.out.println("vlanVal = " + vlanVal);

            ConfPath kp = new ConfPath(vlan.getKeyPath());

            NavuList managedDevices = root.container("devices").list("device");

            for(NavuContainer deviceContainer : managedDevices.elements()){

                NavuContainer ifs = deviceContainer.container("config").
                    container("r", "sys").container("interfaces");

                NavuContainer unit = null;
                if (unitVal != null) {
                    NavuContainer iface =
                        ifs.list("interface").sharedCreate(
                            vlan.leaf("iface").value());
                    iface.leaf("enabled").sharedCreate();
                    unit = iface.list("unit").sharedCreate(unitVal);
                    unit.leaf("description").sharedSet(
                        vlan.leaf("description").value());

                }
                if (unitVal != null && vlanVal != null) {
                    unit.leaf("vlan-id").sharedSet(vlanVal);
                    unit.leaf("enabled").sharedSet(new ConfBool(true));
                    for (ConfValue arpValue : vlan.leafList("arp")) {
                        unit.leafList("arp").sharedCreate(arpValue);
                    }
                }

            }
        } catch (Exception e) {
            throw new DpCallbackException("Could not instantiate service", e);
        } finally {
            try {
                operCtx.finishClearTrans();
            } catch (Exception ignore) {}
        }
        return opaque;
    }

}
```

A Simple Traditional RFM CDB Subscriber

```
public class ConfigCdbSub implements ApplicationComponent {
    private static Logger LOGGER = Logger.getLogger(ConfigCdbSub.class);

    private CdbSubscription sub = null;
    private CdbSession wsess;

    public ConfigCdbSub() {
    }

    @Resource(type=ResourceType.CDB, scope=Scope.CONTEXT,
            qualifier="reactive-fm-loop-subscriber")
    private Cdb cdb;

    @Resource(type=ResourceType.CDB, scope=Scope.CONTEXT,
            qualifier="w-reactive-fm-loop")
    private Cdb wcdb;

    @Resource(type=ResourceType.MAAPI, scope=Scope.INSTANCE,
            qualifier="reactive-fm-m")
    private Maapi maapi;

    public void init() {
        try {
            wsess = wcdb.startSession(CdbDBType.CDB_OPERATIONAL)
            maapi.startUserSession("admin",
                                   InetAddress.getByName("localhost"),
                                   "system",
                                   new String[] {"admin"},
                                   MaapiUserSessionFlag.PROTO_TCP);

            sub = cdb.newSubscription();
            int subid = sub.subscribe(1, new allocVlanService(),
                                      "/avl:alloc-vlan-data");
            // tell CDB we are ready for notifications
            sub.subscribeDone();
        }
        catch (Exception e) {
            LOGGER.error("", e);
        }
    }

    public void run() {
        try {
            while(true) {
                int[] points= sub.read();
                EnumSet<DiffIterateFlags> enumSet =
                    EnumSet.<DiffIterateFlags>of(
                        DiffIterateFlags.ITER_WANT_PREV,
                        DiffIterateFlags.ITER_WANT_ANCESTOR_DELETE,
                        DiffIterateFlags.ITER_WANT_SCHEMA_ORDER);
                ArrayList<Request> reqs = new ArrayList<Request>();
                try {
                    sub.diffIterate(points[0],
                                    new Iter(sub),
                                    enumSet, reqs);
                }
                catch (Exception e) {
                    reqs = null;
                }
                sub.sync(CdbSubscriptionSyncType.DONE_PRIORITY);
```

```
                for (Request req : reqs) {
                    if ((req.op == Operation.CREATE) &&
                        (req.t == Type.UNIT)) {
                     int unit = alloc_unit();

                     System.out.println("SET: " + req.path + "/unit -> " + unit);
                    wsess.setElem(new ConfBuf(unit+""), req.path + "/unit");

                    redeploy("/avl:alloc-vlan{%x}/reactive-re-deploy",
                             req.key, maapi);
                    }

                    else if ((req.op == Operation.CREATE) &&
                             (req.t == Type.VID)) {

                    int vid = alloc_vid();

                    System.out.println("SET: " + req.path + "/vln-id -> " + vid);
                    wsess.setElem(new ConfUInt16(vid), req.path + "/vlan-id");

                    redeploy("/avl:alloc-vlan{%x}/reactive-re-deploy",
                             req.key, maapi);
                    }
                    else if (req.op == Operation.DELETE) {

                      try {
                        ConfValue v = wsess.getElem(req.path + "/unit");
                        deallocate_unit(v);
                        wsess.delete(req.path + "/unit");
                      } catch (Exception e) {
                          ;
                      }
                      try {
                        ConfValue v = wsess.getElem(req.path + "/vlan-id");
                        deallocate_vid(v);
                        wsess.delete(req.path + "/vlan-id");
                      } catch (Exception e) {
                          ;
                      }
                    }

                }
            }
        } catch (SocketException e) {
          }
        catch (Exception e) {
            LOGGER.error("",e );
        }
    }

    public void finish() {
        safeclose(cdb);
        safeclose(wcdb);
        try {
            maapi.getSocket().close();
        } catch (Exception e) {
        }
    }
```

```
    private void safeclose(Cdb s) {
        try {s.close();}
        catch (Exception ignore) {}
    }

    private enum Operation { CREATE, DELETE}
    private enum Type { UNIT, VID}

    private class Request {
        ConfKey key;
        Operation op;
        Type t;
        ConfPath path;
    }

    private class Iter implements CdbDiffIterate {
        CdbSubscription cdbSub;

        Iter(CdbSubscription sub ) {
            this.cdbSub = sub;
        }

        public DiffIterateResultFlag iterate(
            ConfObject[] kp,
            DiffIterateOperFlag op,
            ConfObject oldValue,
            ConfObject newValue, Object initstate) {

            ArrayList<Request> reqs = (ArrayList<Request>) initstate;

            try {
                ConfPath p = new ConfPath(kp);
                System.out.println("ITER " + op + " " + p);
                ConfKey key = (ConfKey) kp[kp.length-2];
                Request r = new Request();
                r.path = p; r.key = key;

                if ((op == DiffIterateOperFlag.MOP_CREATED) &&
                    kp[0].toString().equals("avl:request-allocate-unit")) {
                    r.op = Operation.CREATE; r.t = Type.UNIT;
                    reqs.add(r);
                } else if ((op == DiffIterateOperFlag.MOP_CREATED) &&
                           kp[0].toString().equals("avl:request-allocate-vid")) {
                    r.op = Operation.CREATE; r.t = Type.VID;
                    reqs.add(r);
                } else if ((op == DiffIterateOperFlag.MOP_DELETED)) {
                    r.op = Operation.DELETE;
                    reqs.add(r);
                }
            } catch (Exception e) {
                LOGGER.error("", e);
            }
            return DiffIterateResultFlag.ITER_RECURSE;
        }
    }

    private int[] units = null;
    private int[] vids = null;
```

```
    private int alloc_unit() {
        if (units == null) {
            units = new int[256];
            for (int i = 0; i < 256; i++)
                units[i]=-1;
        }
        for (int i =0; i < 256; i++) {
            if (units[i] == -1) {
                units[i] = i;
                return i;
            }
        }
        return -1;
    }
    private void deallocate_unit(ConfValue v) {
        int i = Integer.parseInt(v.toString());
        units[i] = -1;
    }
    private int alloc_vid() {
        if (vids == null) {
            vids = new int[256];
            for (int i =0; i < 256; i++)
                vids[i]=-1;
        }
        for (int i =0; i < 256; i++) {
            if (vids[i] == -1) {
                vids[i] = i;
                return i;
            }
        }
        return -1;
    }
    private void deallocate_vid(ConfValue v) {
        long i = ((ConfUInt16)v).longValue();
        vids[(int)i] = -1;
    }
    private void redeploy(String path, ConfKey k, Maapi m) {
        Redeployer r = new Redeployer(path, k, m);
        Thread t = new Thread(r);
        t.start();
    }
    private class Redeployer implements Runnable {
        private String path;
        private ConfKey k;
        private Maapi m;

        public Redeployer(String path, ConfKey k, Maapi m) {
            this.path = path; this.k = k; this.m = m;
        }
        public void run() {
            try {
                m.requestAction(new ConfXMLParam[] {},
                                "/avl:alloc-vlan{%x}/reactive-re-deploy",
                                k);
            } catch (Exception e) {
                throw new RuntimeException("error in reactive-re-deploy", e);
            }
        }
    }
}
```

What are the RFM shortcomings?

Works very well for staged creates.

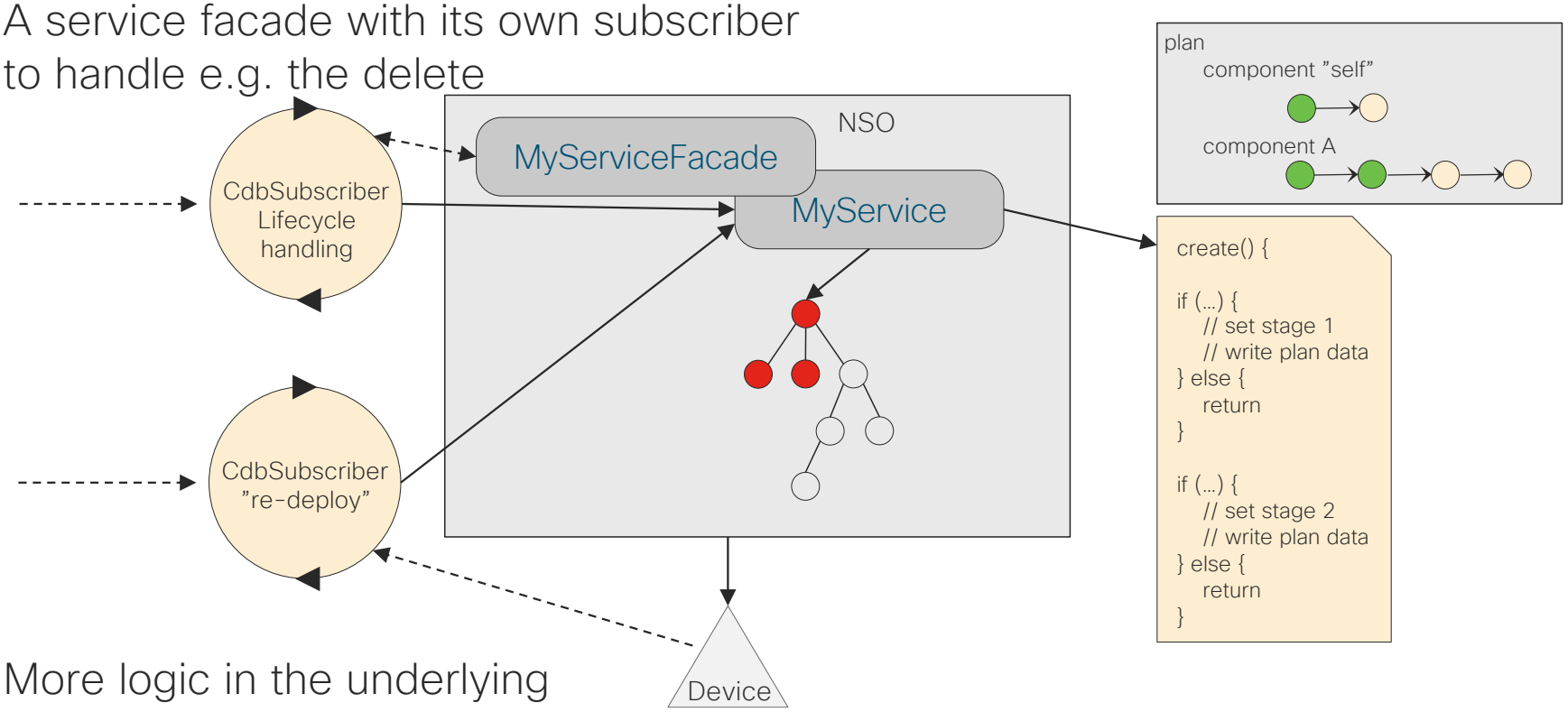
Doing staged updates or deletes are increasingly difficult.

RFM with complex lifecycle.

- More logic has to be added to the CdbSubscriber to handle different lifecycle event.
- At delete, a service is just deleted. So staging implies having a façade service which can be deleted while the underlaying service is deleting (updating with less and less written config).
- Some lifecycle cases are easily missed in the code.

RFM with complex lifecycle (staged delete)

A service facade with its own subscriber to handle e.g. the delete



More logic in the underlying subscriber to handle lifecycle events

Why are Nano Services needed?

Reactive FastMap (RFM) is the driving force.

Native support for all lifecycle events with a deterministic lifecycle

Keeping service code to a minimum and located in one place.

Nano service Code example

The CDB subscriber is replaced by YANG declarations.

A Simple Nano Service

```
public class AllocVlanServiceRFS {
    private static Logger LOGGER Logger.getLogger(AllocVlanServiceRFS.class);
    private NavuContext operCtx = null;

    @NanoServiceCallback(servicePoint="alloc-vlanspnt",
                         componentType="avi:vlan-iface", state="ncs:init",
                         callType=NanoServiceCBType.CREATE)
    public Properties vlanIfaceInitCreate(NanoServiceContext context,
                                  NavuNode vlanService,
                                  NavuNode root,
                                  Properties opaque,
                                  Properties componentProperties)
                                  throws DpCallbackException {
        System.out.println("vlan-iface alloc-req CREATE()");
        LOGGER.setLevel(Level.ALL);
        checkThatSyncFromHasBeenPerformed(root);

        try {
            ConfValue serviceKey = vlanService.leaf("name").value();
            LOGGER.debug("ServiceKey = "+ serviceKey.toString());
            NavuList transRequestList = getTransRequestList(root);
            NavuContainer request = transRequestList.sharedCreate(serviceKey);
            request.leaf("owner").sharedSet(
                new ConfPath("/avi:alloc-vlan{%s}", serviceKey).toString());
        } catch (Exception e) {
            throw new DpCallbackException("Could not instantiate service", e);
        }
        return opaque; // opaque is not used by this service
    }

    @NanoServiceCallback(servicePoint="alloc-vlanspnt",
                         componentType="avi:vlan-iface", state="avi:vid-alloc",
                         callType=NanoServiceCBType.CREATE)
    public Properties vlanIfaceAllocCreate(NanoServiceContext context,
                                  NavuNode vlanService,
                                  NavuNode root,
                                  Properties opaque,
                                  Properties componentProperties)
                                  throws DpCallbackException {
        System.out.println("vlan-iface vid-alloc CREATE()");
        LOGGER.setLevel(Level.ALL);
        try {
            ConfValue serviceKey = vlanService.leaf("name").value();
            LOGGER.debug("ServiceKey = "+ serviceKey.toString());
            NavuList runningRequestList = getRunningRequestList(root);
            NavuContainer runningRequest = runningRequestList
                .elem(serviceKey.toString());
            ConfUInt32 unit = getChildLeafUInt32Value(runningRequest, "unit");
             ConfUInt32 vlan = getChildLeafUInt32Value(runningRequest, "vlan");
            if (opaque == null) {
                opaque = new Properties();
            }
            opaque.setProperty("UNITVAL", "" + unit.toString() + "");
            writeDeviceConfig(root, vlanService, unit, vlan);
        } catch (Exception e) {
            throw new DpCallbackException("Could not instantiate service", e);
        }
        return opaque; // opaque is not used by this service
    }
```

```
    private void writeDeviceConfig(NavuNode root, NavuNode vlanService,
                                   ConfUInt32 unitVal, ConfUInt32 vlanVal) throws NavuException {
        NavuList managedDevices = root.container("devices").list("device");
        for(NavuContainer deviceContainer : managedDevices.elements()) {
            NavuContainer ifs = deviceContainer.container("config").
                container("r", "sys").container("interfaces");
            NavuContainer unit = null;
            if (unitVal != null) {
                NavuContainer iface = ifs.list("interface").sharedCreate(vlanService.leaf("iface").value());
                iface.leaf("enabled").sharedCreate();
                unit = iface.list("unit").sharedCreate(new ConfUInt32((int)unitVal.longValue()));
                unit.leaf("description").sharedSet(vlanService.leaf("description").value());
            }
            if (unitVal != null && vlanVal != null) {
                unit.leaf("vlan-id"). sharedSet(new ConfUInt16(vlanVal.longValue()));
                unit.leaf("enabled").sharedSet(new ConfBool(true));
                for (ConfValue arpValue : vlanService.leafList("arp")) {
                    unit.leafList("arp").sharedCreate(arpValue);
                }
            }
        }
    }

    private ConfUInt32 getChildLeafUInt32Value(NavuContainer runningRequest,
                                               String leafname) throws NavuException {
        if(runningRequest != null) {
            NavuLeaf runningLeaf = runningRequest.leaf(leafname);
            if (runningLeaf != null) return(ConfUInt32) runningLeaf.value();
        }
        return null;
    }

    private NavuList getTransRequestList(NavuNode root) throws NavuException {
        return root.getParent().container(xcimrm.hash). container("xcimrm").list("request");
    }

    private NavuList getRunningRequestList(NavuNode root) throws NavuException {
        startReadTransRunningOper(root);
        NavuContainer operBase = new NavuContainer(this.operCtx);
        NavuContainer operRoot = operBase.container(xcimrm.hash);
]       return operRoot.container(xcimrm.prefix, "xcimrm").list("request");
    }

    private int startReadTransRunningOper(NavuNode root) throws NavuException {
        this.operCtx = new NavuContext(root.context().getMaapi());
        int th = operCtx.startOperationalTrans(Conf.MODE_READ);
        return th;
    }

    private void checkThatSyncFromHasBeenPerformed(NavuNode root) throws Exception {
        NavuList managedDevices = root. container("devices").list("device");
        for (NavuContainer device : managedDevices) {
            if (device.list("capability").isEmpty()) {
                String mess = "Device %1$s has no capabilities, has sync-from been performed?";
                String key = device.getKey().elementAt(0).toString();
                throw new DpCallbackException(String.format(mess, key));
            }
        }
    }
}
```

Simple Nano Service YANG statements

```
ncs:plan-outline vlan-plan {
  description "Service alloc-vlan plan definition";

  ncs:component-type "ncs:self" {
    ncs:state "ncs:init";
    ncs:state "ncs:ready" {
      ncs:nano-callback;
    }
  }

  ncs:component-type "avl:vlan-iface" {
    ncs:state "ncs:init" {
      ncs:post-action-node "/ncs:devices" {
        ncs:action-name "sync-from";
        ncs:result-expr "count(sync-result/result) = 1";
      }
    }
    ncs:state "avl:unit-alloc" {
      ncs:create {
        ncs:pre-condition {
          ncs:monitor
            "/myserv:myoper[name=$SERVICE/name]" {
            ncs:trigger-expr "syslog = 'true'";
          }
        }
      }
    }
    ncs:state "avl:vid-alloc" {
      ncs:create {
        ncs:nano-callback;
        ncs:pre-condition {
          ncs:monitor
            "/myserv:myoper[name=$SERVICE/name]" {
            ncs:trigger-expr "syslog = 'true'";
          }
        }
      }
    }
    ncs:state "ncs:ready";
  }
}

ncs:service-behavior-tree alloc-vlanspnt {
  description
    "Behaviour tree for alloc-vlan service";

  ncs:plan-outline-ref vlan-plan;

  ncs:selector {
    ncs:create-component "'self'" {
      ncs:component-type-ref "ncs:self";
    }
    ncs:create-component "vlan" {
      ncs:component-type-ref "avl:vlan-iface";
    }
  }
}
}
```

Nano vs Traditional Services.

The original FastMap algorithm is indifferent to ”staging”. Needs external means to re-deploy.

A Nano service executes on the plan and has an enhanced FastMap algorithm that supports “staging”. Re-deploys natively using kickers.

What is a kicker?

A kicker monitors a data node (config or oper) and can call an action when the data node changes.

Simpler than a CDB subscriber.

Created by writing kicker declarations in a transaction and takes effect after this transaction is committed.

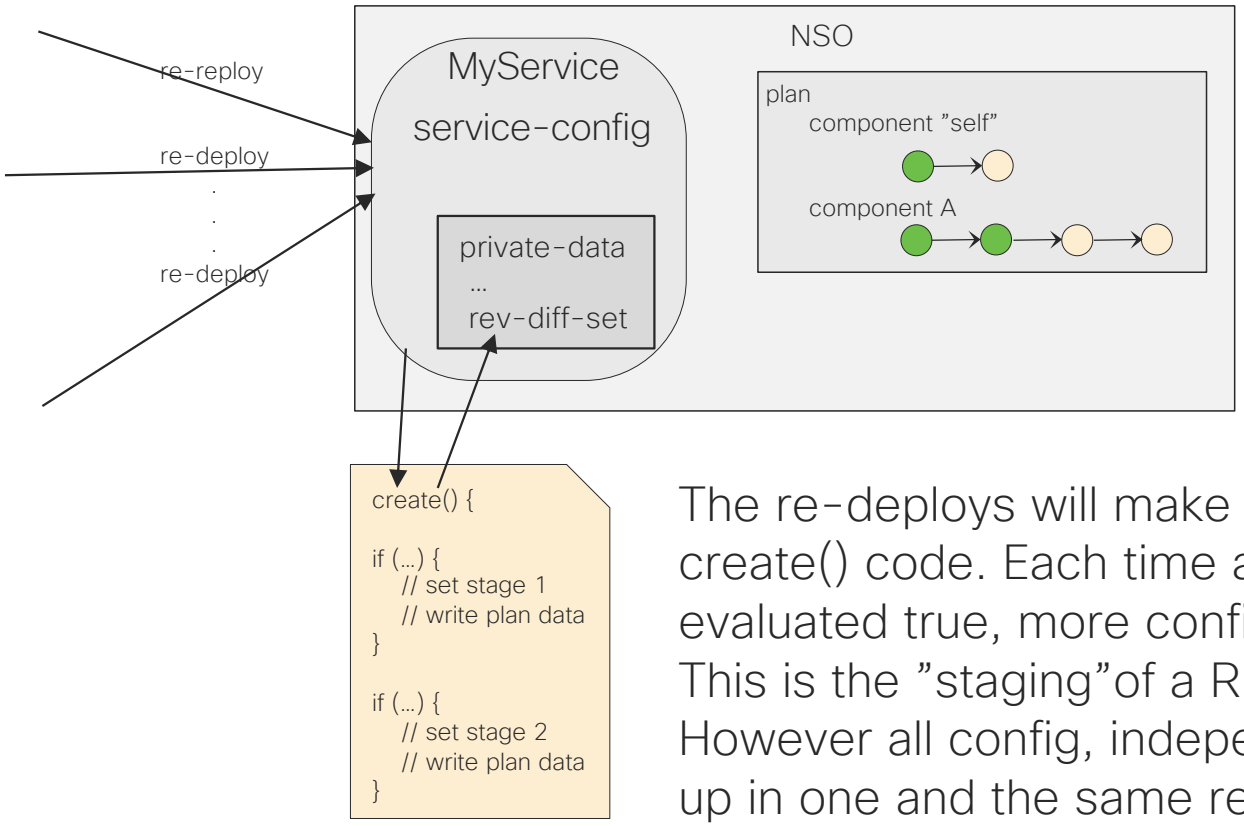
Note, does not work on data written using the CDB API.

Internals of Nano vs Traditional Services.

The are as YANG declarations the same type of “servicepoint”.

Any actions or operational data on the service are still valid

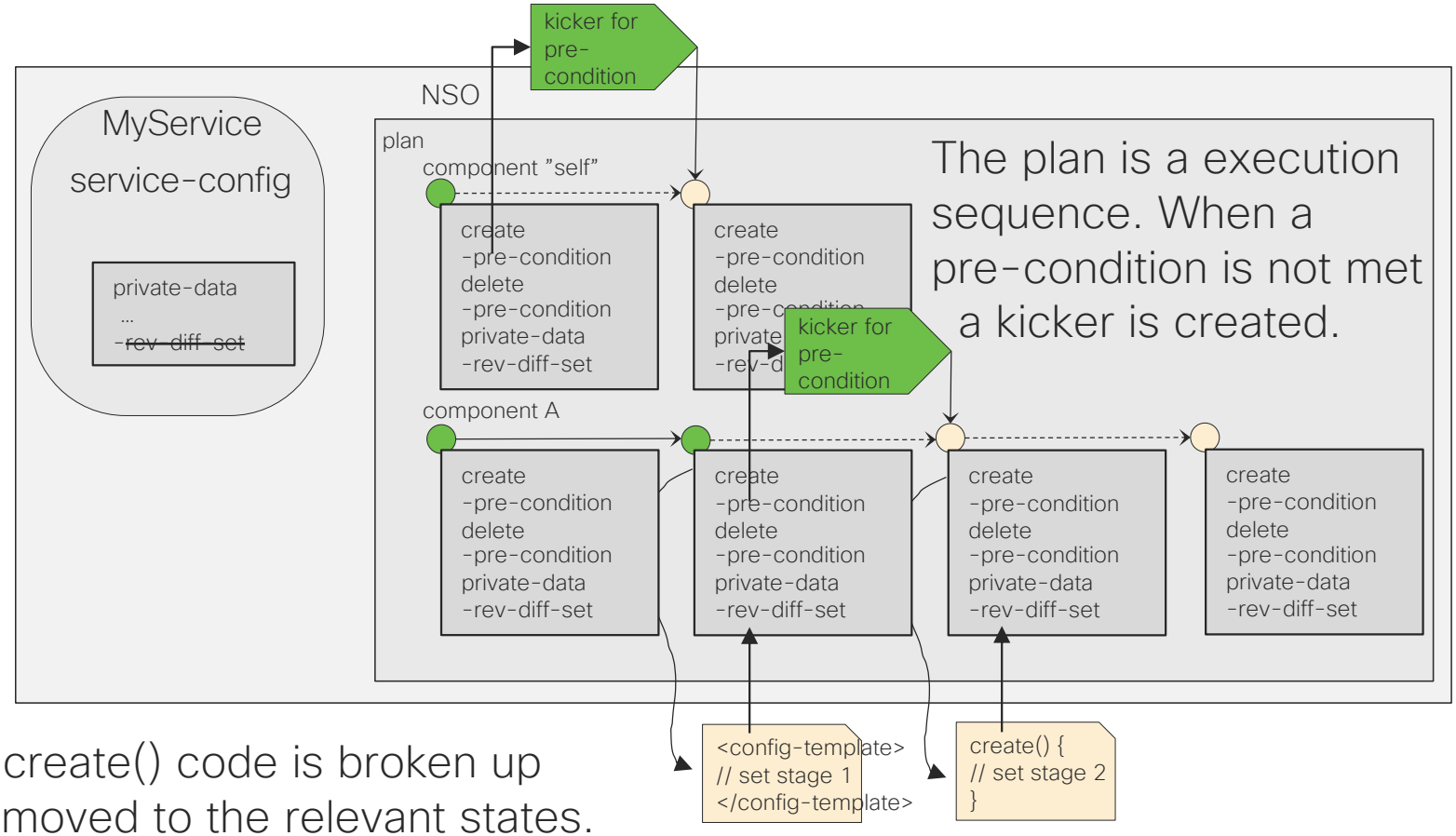
Internals of a traditional service



The plan is a log that reflects the current state of the RFM service.

The re-deploys will make the service re-enter the create() code. Each time an new "if" clause is evaluated true, more config is set on the devices. This is the "staging"of a RFM.
However all config, independent of stage, is ending up in one and the same rev-diff-set

Internals of a Nano service




```
create() {

if (...) {
  // set stage 1
  // write plan data
}

if (...) {
  // set stage 2
  // write plan data
}
```

The create() code is broken up and moved to the relevant states.

What does the Nano service executable plan imply?

Since the component and its states constitutes a execution order, this order can be controlled and changed.

Typically delete operations should go in reverse order – backtrack

Why would a Nano plan-component backtrack?

- A component is removed from the plan.
- A pre-condition in a components state was previously satisfied but this is no longer the case.
- The complete service instance is deleted.

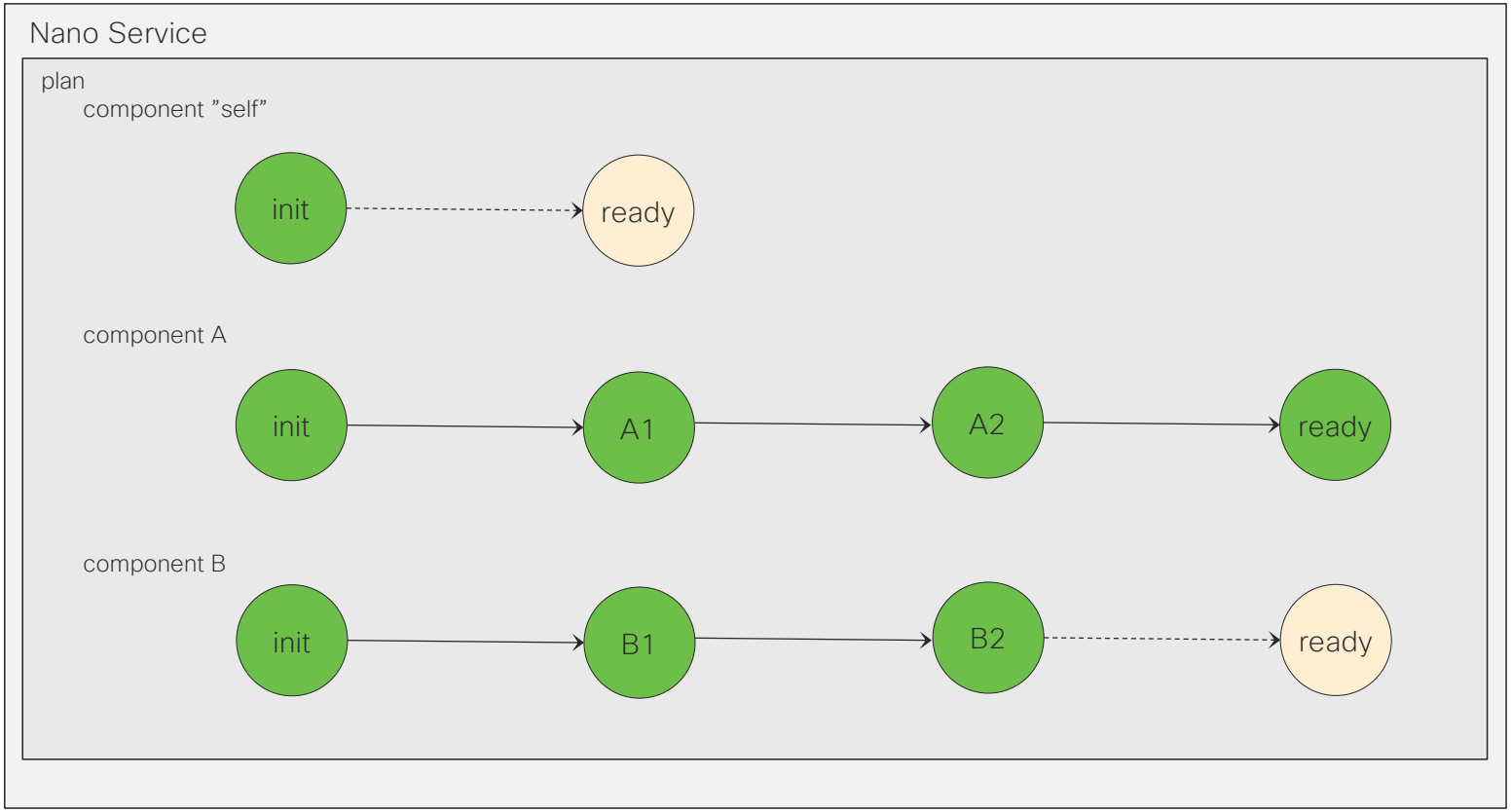
A plan component is removed from the plan

- Plan components can be removed independently of each other
- Removal of config from a component is performed in “reverse” order
- Other components can still progress forward.
- When the component is backtracking is complete the component is removed from the plan.

Scenario 1, initially converging

Initially all components are progressing forward.

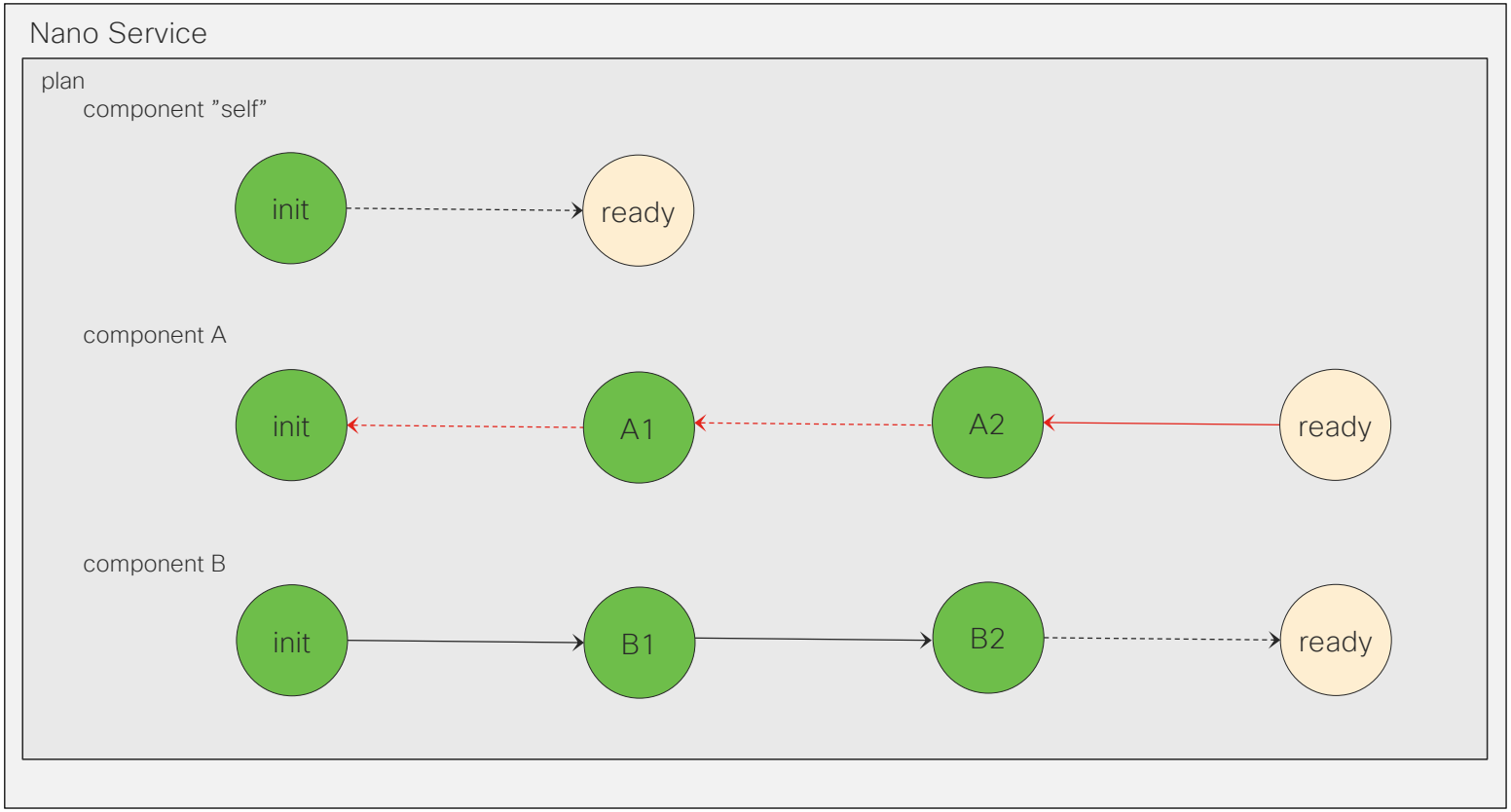
Each component are independent and execution stops at the first pre-condition that evaluates to false.



Scenario 1, Compontent A removed, backtracking

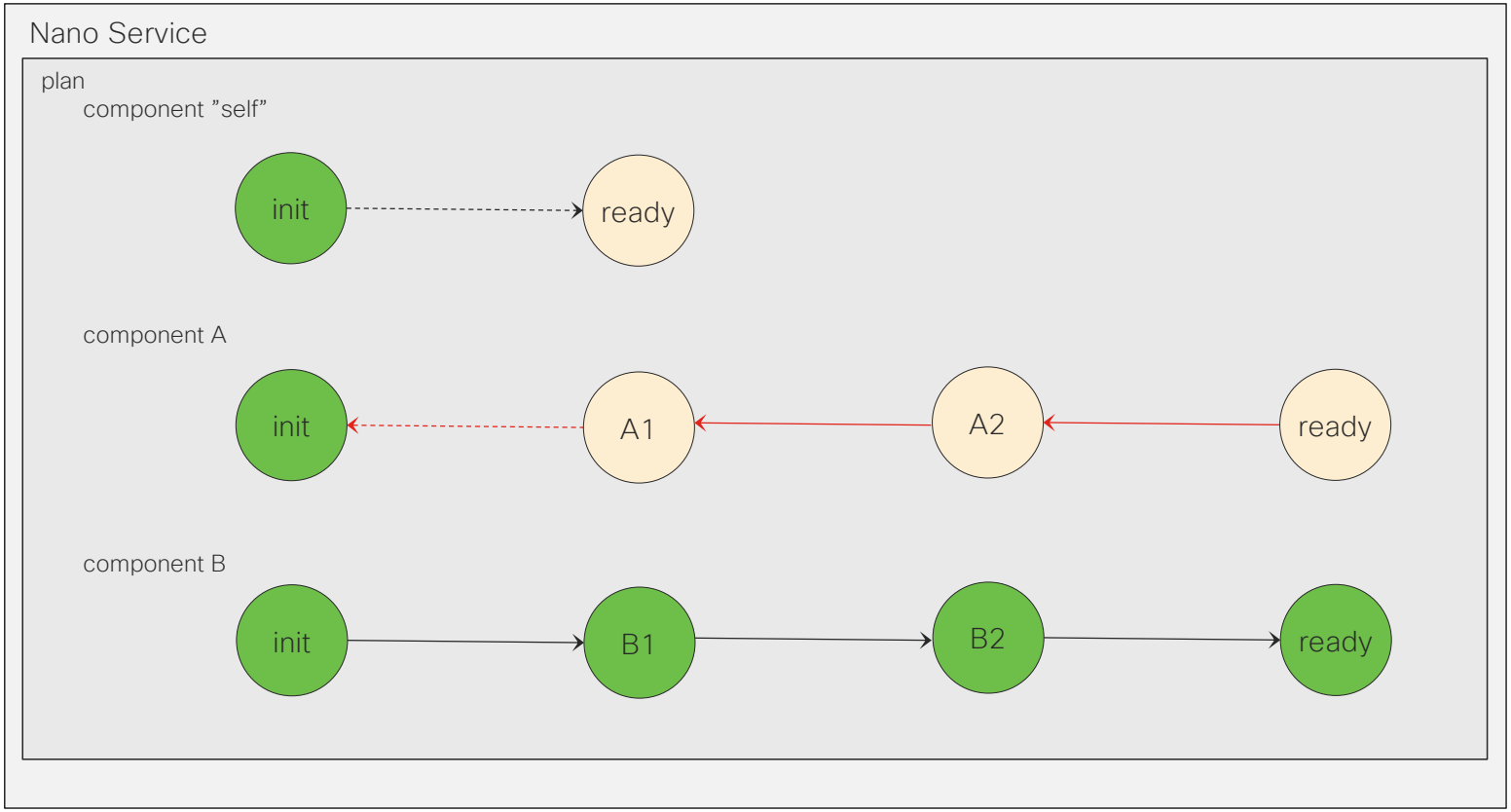
At some point it is decided that a component should be removed.

The component is set to backtracking and device config is removed in the reverse order. Still execution stops At the first delete pre-condition that evaluates to false.



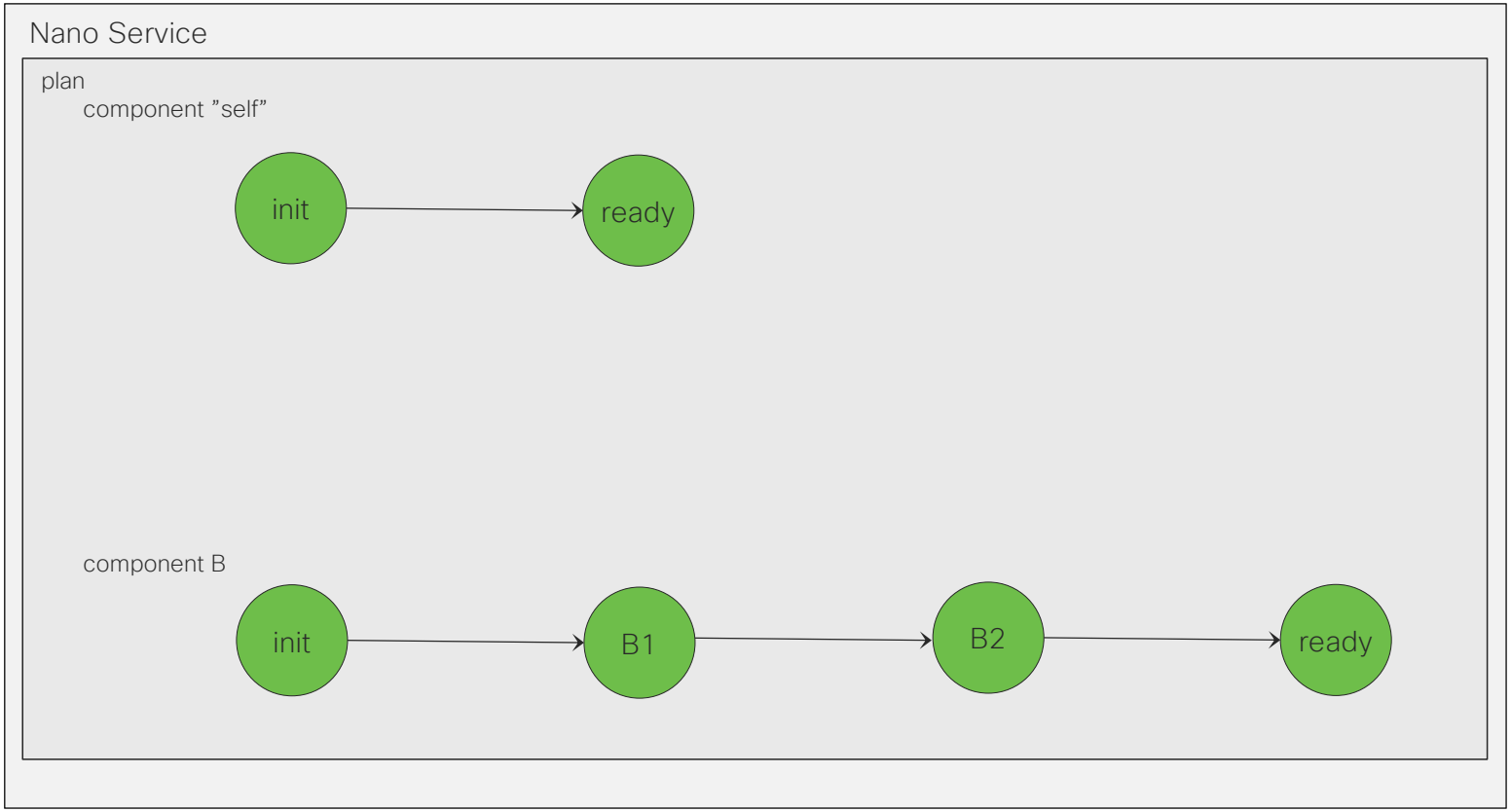
Scenario 1,Parallel backtrack and forward progress

While a component is backtracking other components can still commence their forward execution.



Scenario 1, Remove of backtracked component

When the component is completely backtracked it is removed from the plan.



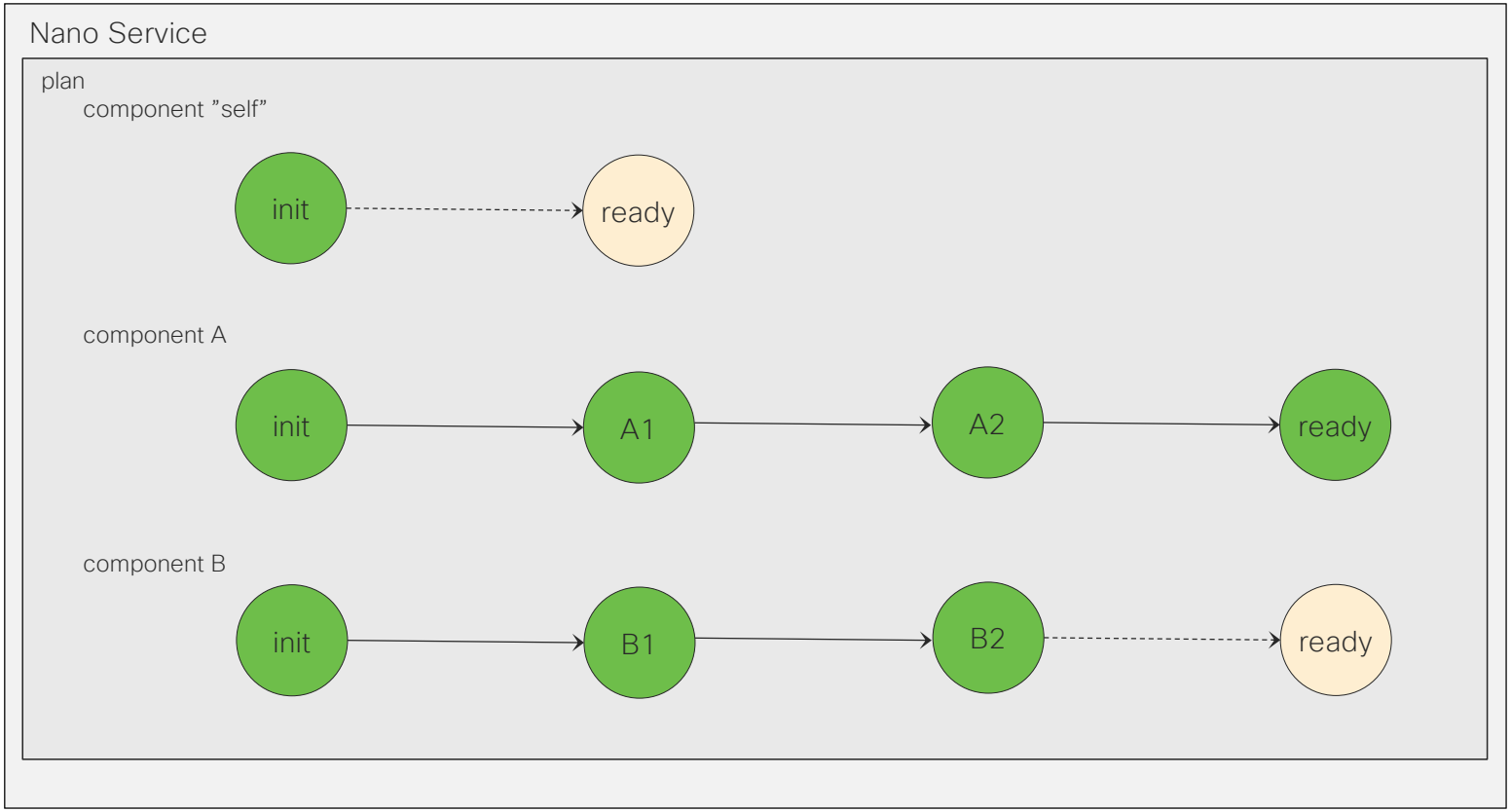
A state pre-condition is no longer satisfied.

- If a state in a plan component was satisfied but this is no longer the case, this implies that the component has progressed “to far”.
- This backtrack motion has in this case a goal state, which is the state that fails to be satisfied
- When the goal state is reached the component switches to forward execution again.

Scenario 2, initially converging

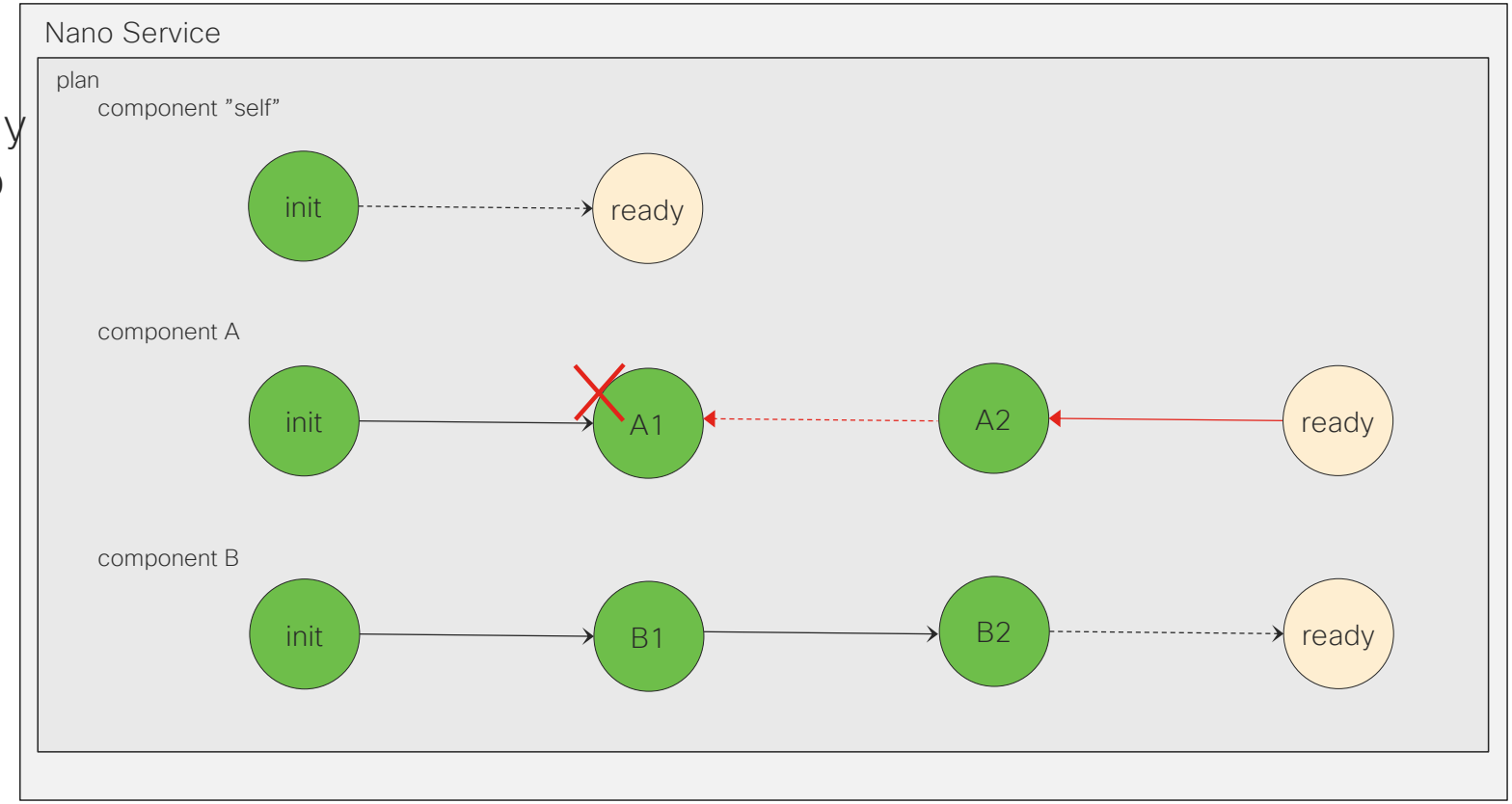
Initially all components are progressing forward.

Any re-deploy of the service expects, and will validate that all previously met pre-conditions are still satisfied.



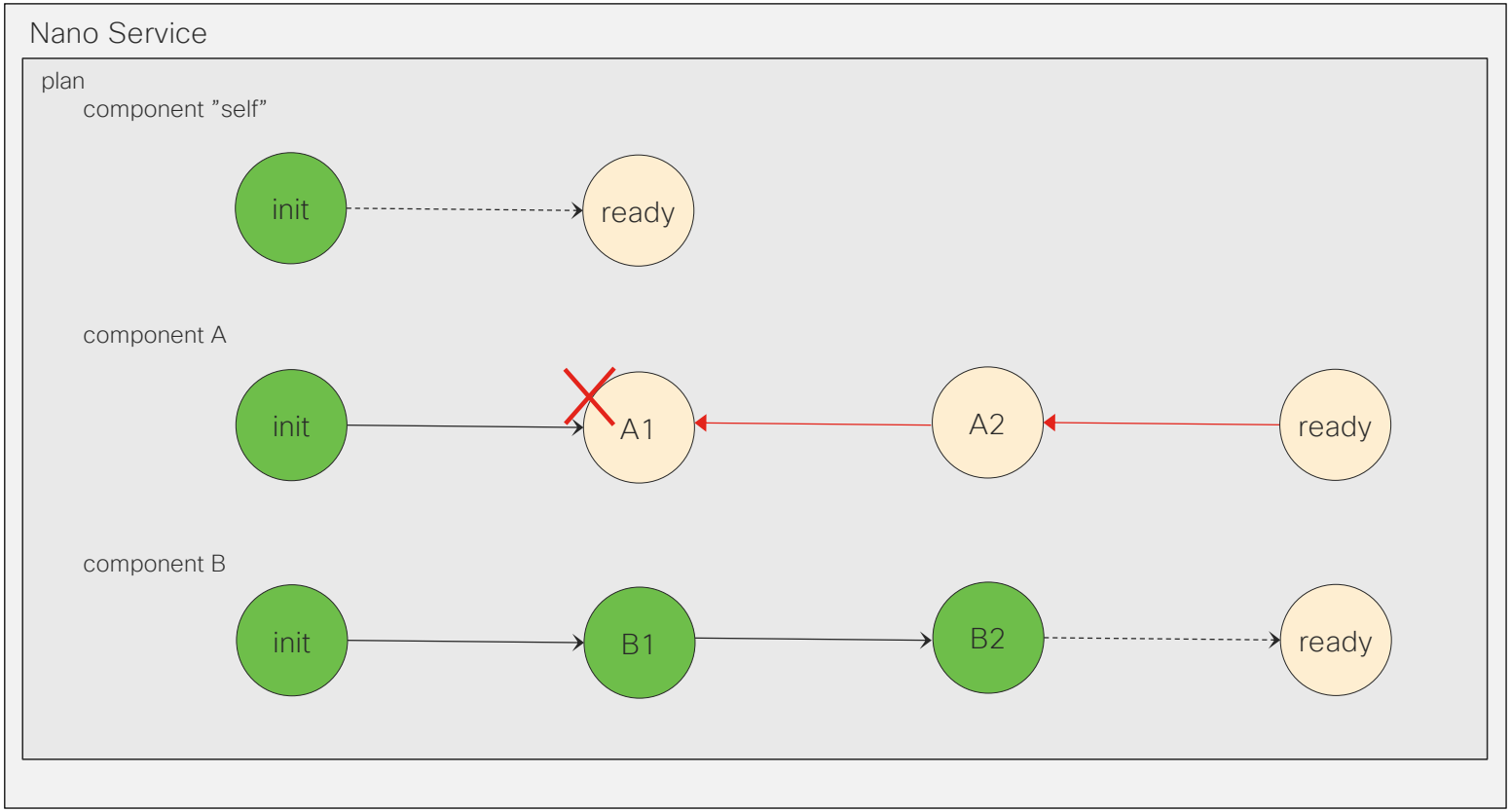
Scenario 1, state pre-condition no longer satisfied

If a re-deploy will identify that previously met pre-condition no longer is satisfied it will immediately start to backtrack that component.



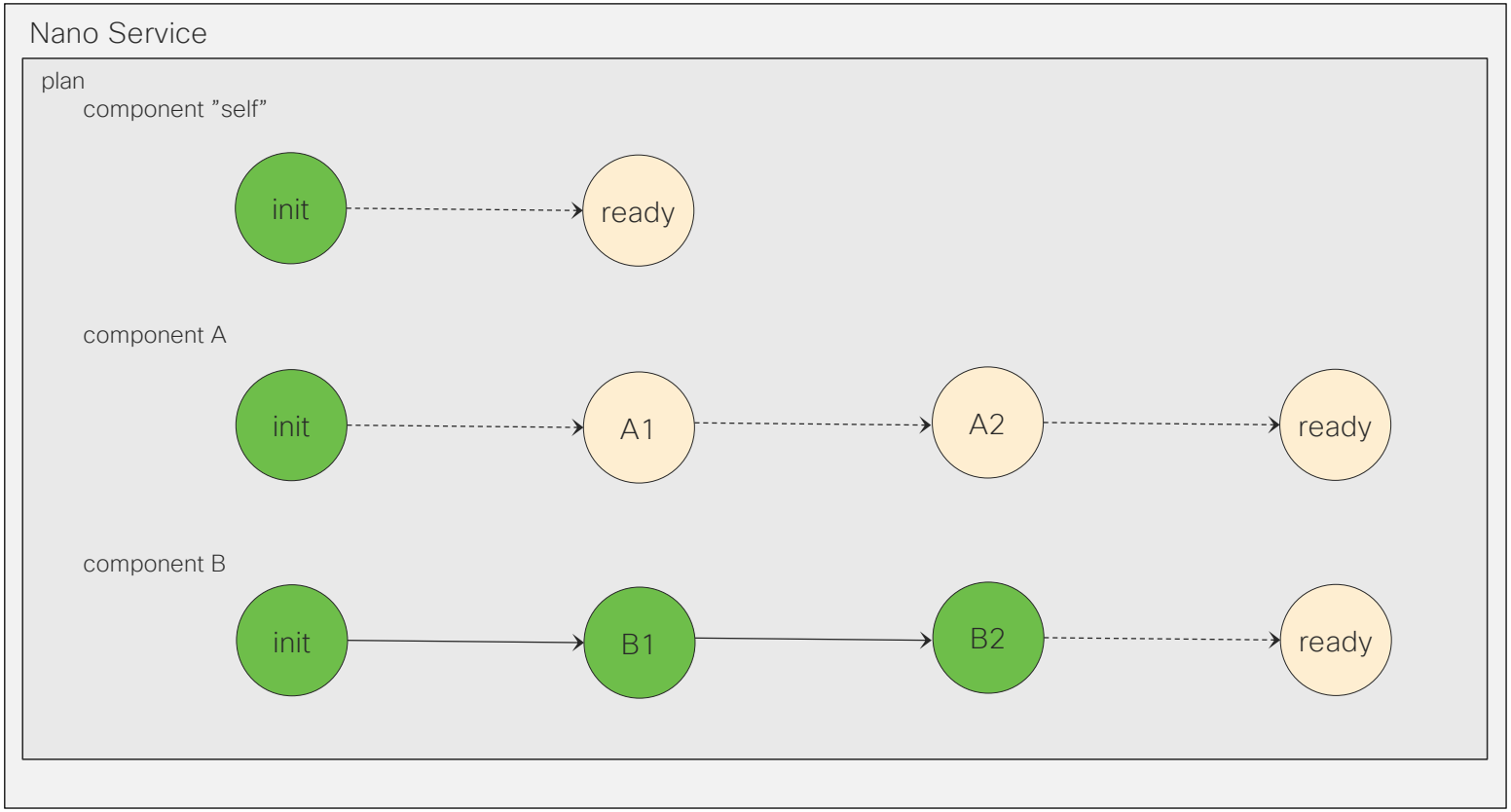
Scenario 1, backtracking to the goal state

The backtracking component will backtrack to the state with the failing pre-condition.



Scenario 2, component switch to forward progress

When the device config is removed up to the state with the failing pre-condition it will switch back to normal forward execution again.



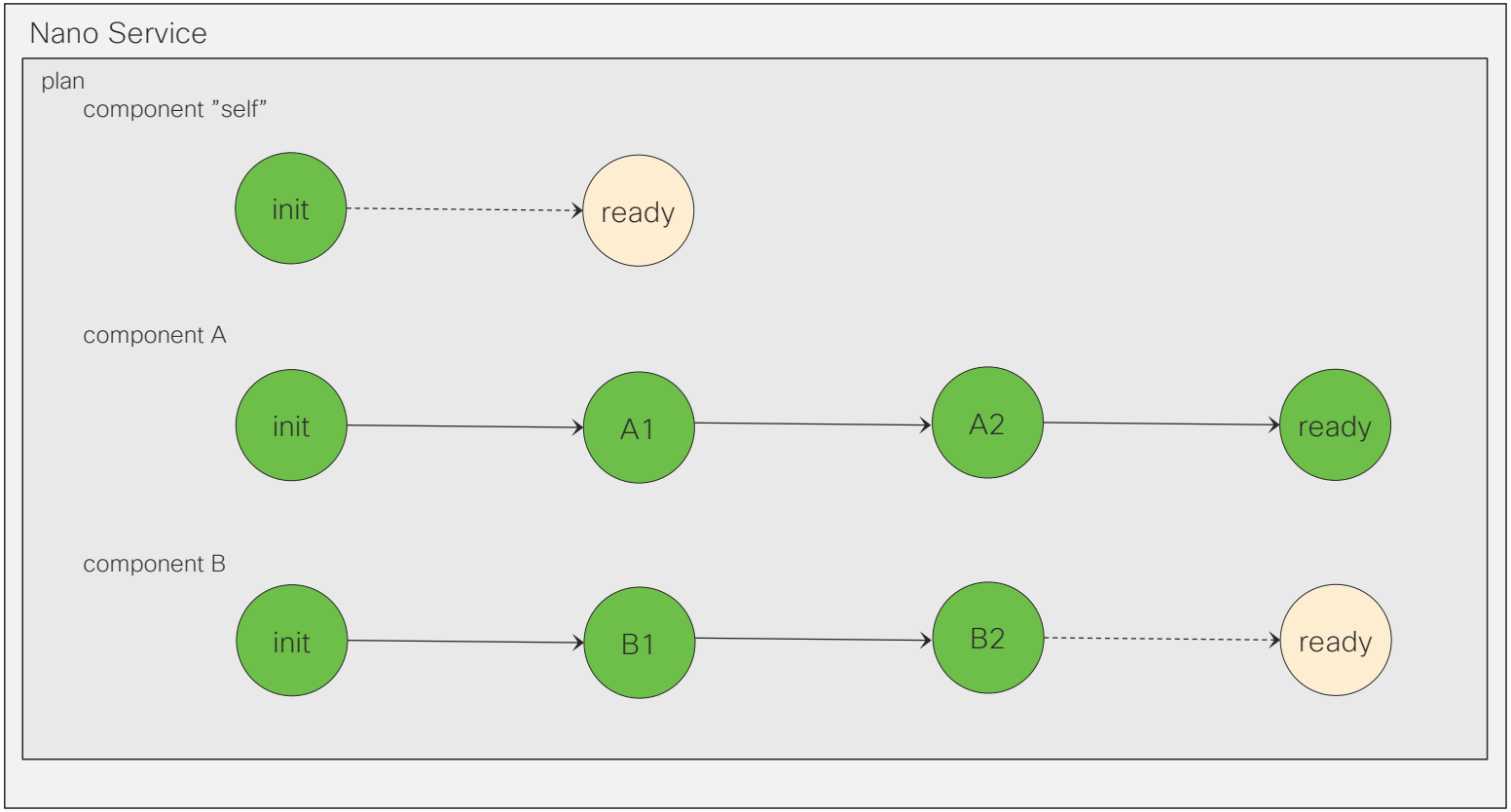
A service is deleted

- If a service is deleted the service parameters together with its executable plan (including rev-diff-sets) is “zombiefied” i.e. stored as a blob in a zombie list.
- The service instance itself is removed but it cannot be recreated while the zombie is backtracking.
- When all components are backtracked and removed the zombie is removed from the list.
- The zombie list contains the plan that shows the backtracking progress.

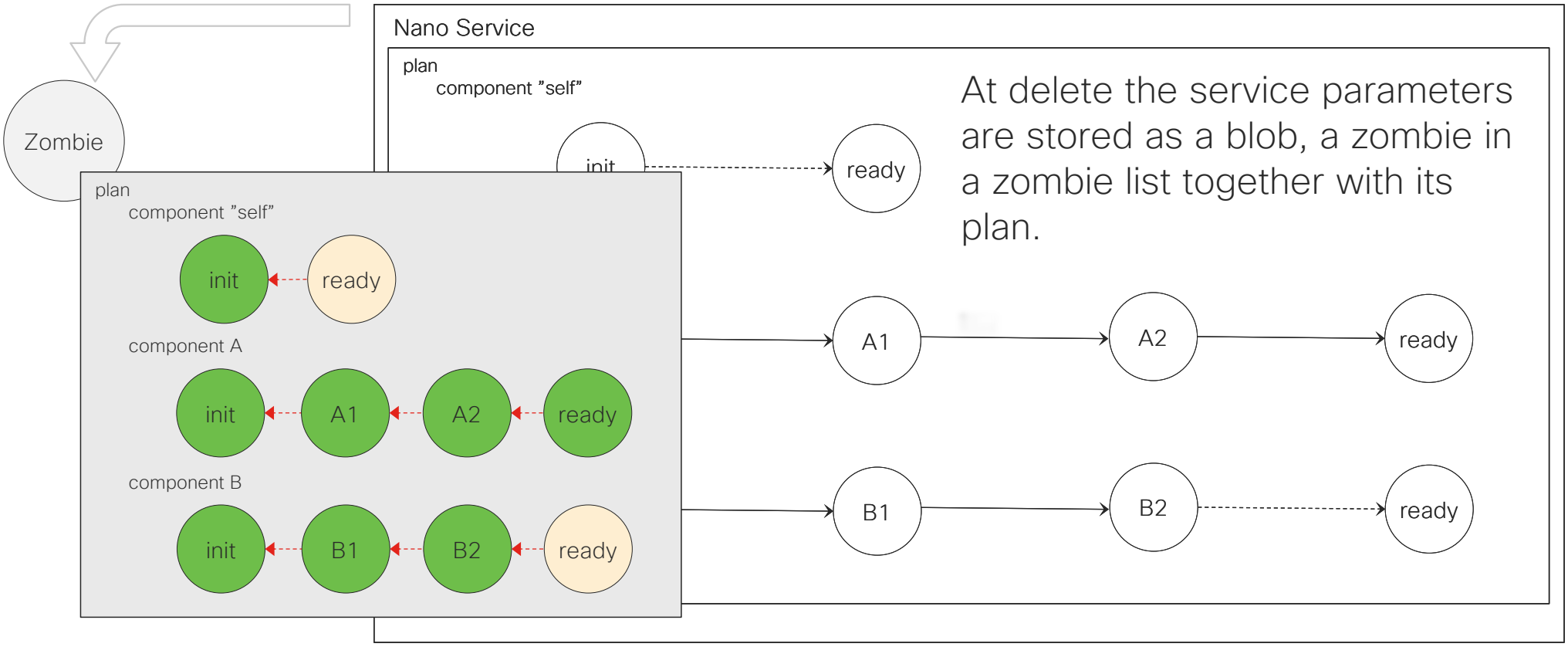
Scenario 3, initially converging

Initially all components are progressing forward.

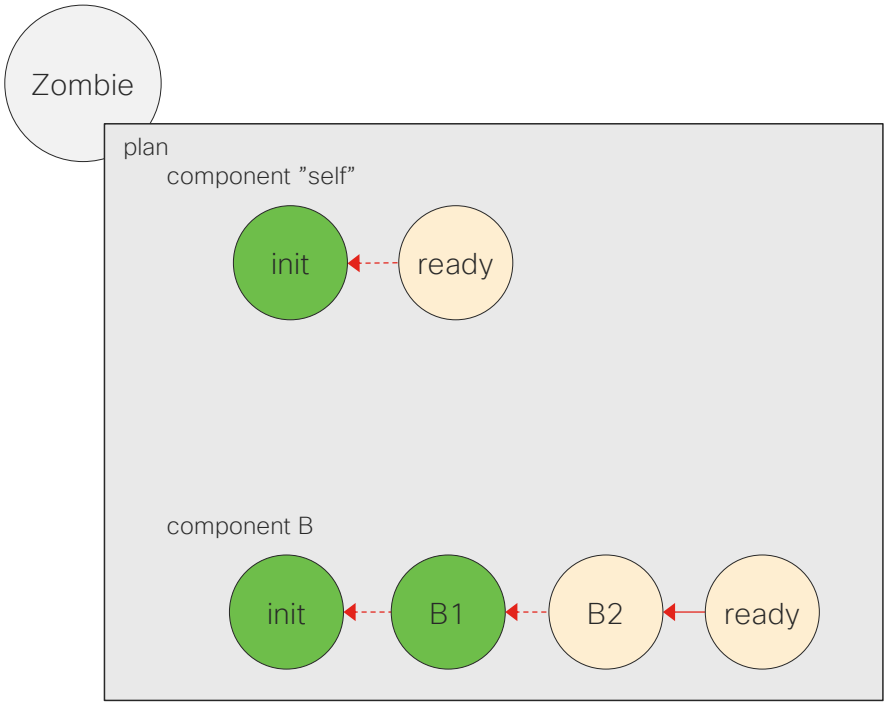
At some point the complete service is deleted.



Scenario 3, Service deleted, zombie created



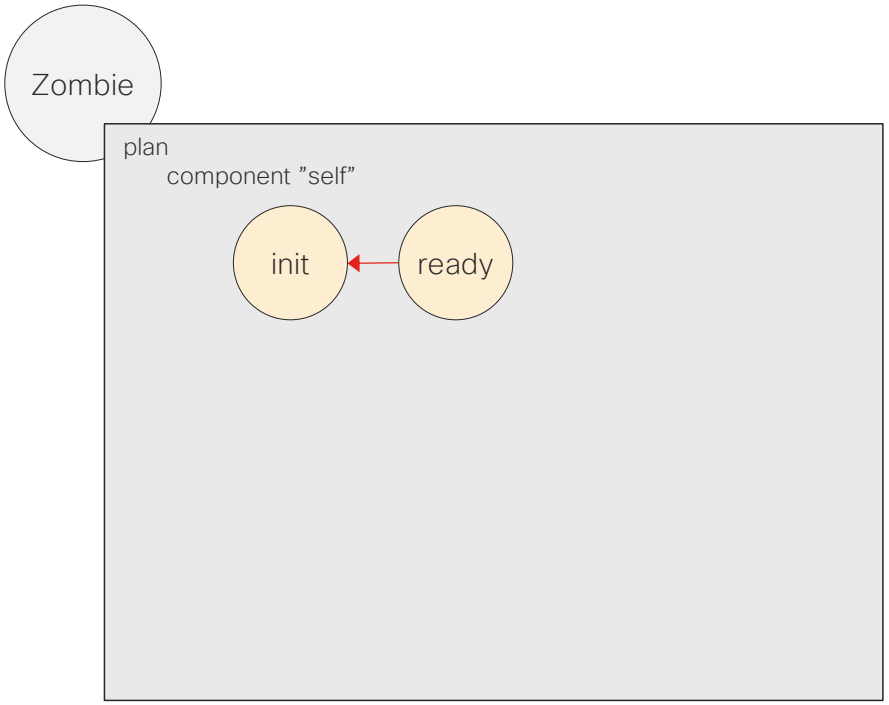
Scenario 3, Zombie service components backtrack



The zombie starts backtracking all of the plan components. The same principles, with delete pre-conditions controlling which states can be removed, holds for the zombie.

During the zombie phase the same, deleted, service instance cannot be recreated.

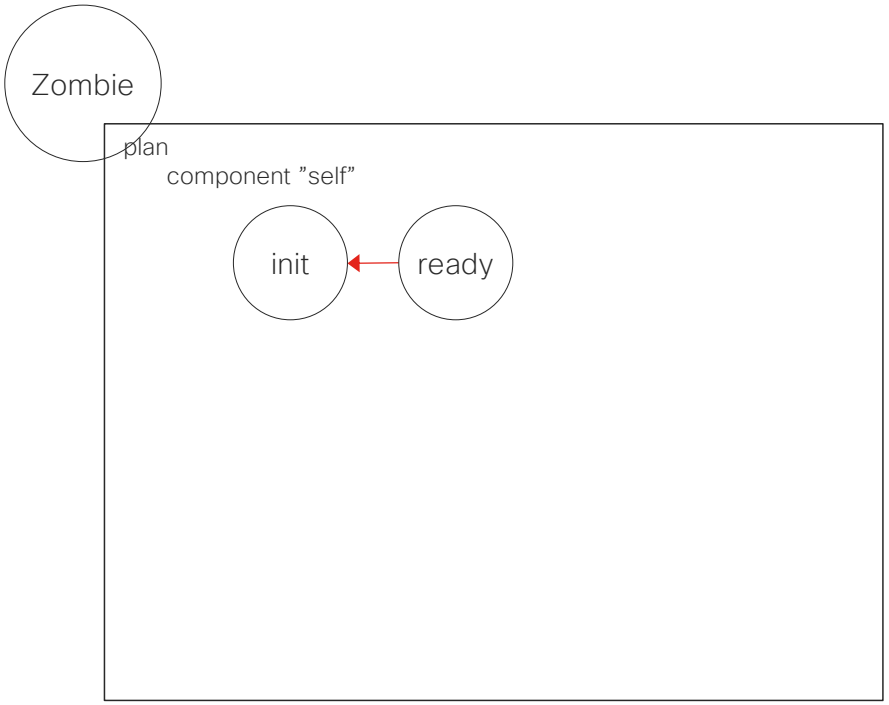
Scenario 3, Zombie components are removed



The last component to be removed from a zombie plan is the self component.

At this point all device config is removed by the service zombie.

Scenario 3, Finally zombie is removed



When all components are removed the zombie itself is removed.

At this point the same service instance as was previously a zombie is now allowed to be recreated, if that would be the case.

How does Nano services handle actions?

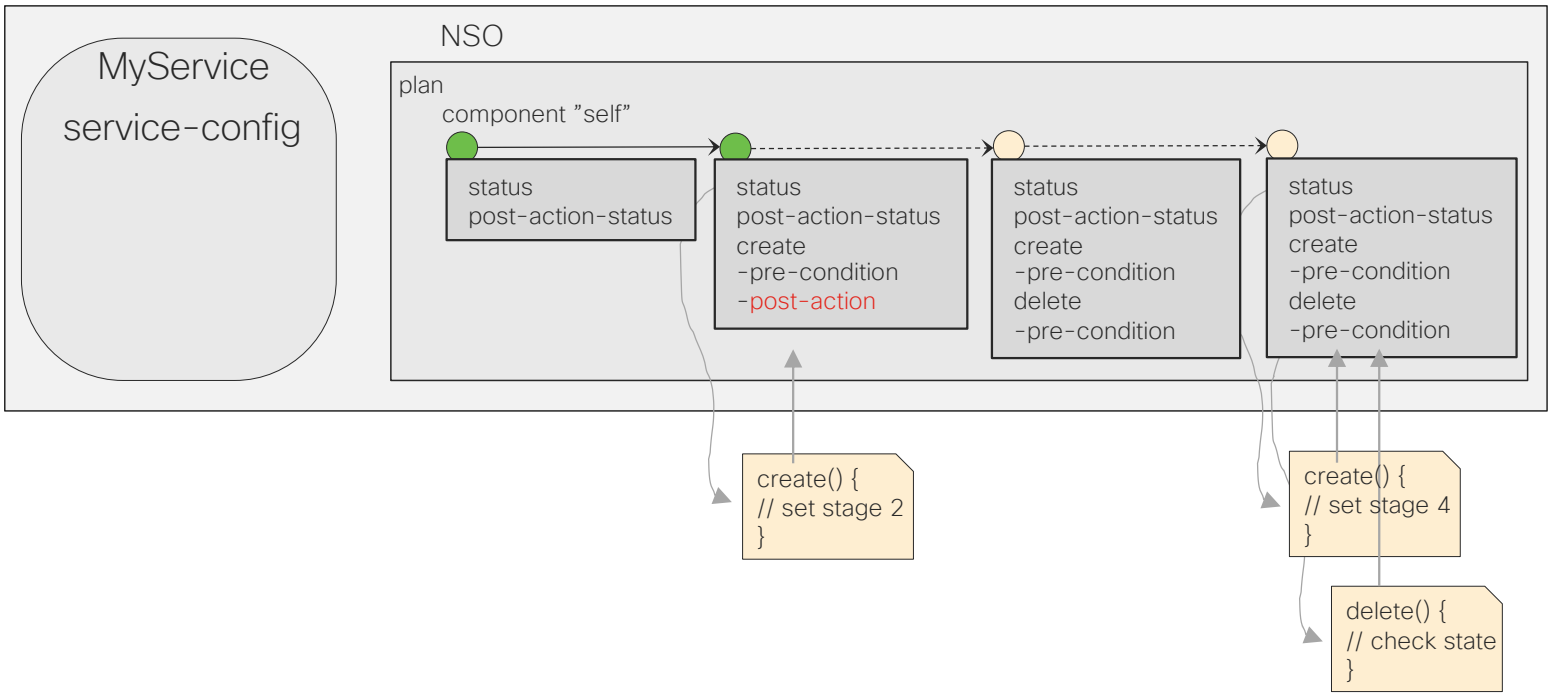
In FastMap any actions, rpcs or time consuming operation must be avoided.

In a RFM these are found in the CDB subscribers. They are often necessary e.g. when an rpc must be called to free the license of a virtual devices that should be deleted etc.

Nano services implements these as state post-actions that can run asynchronously.

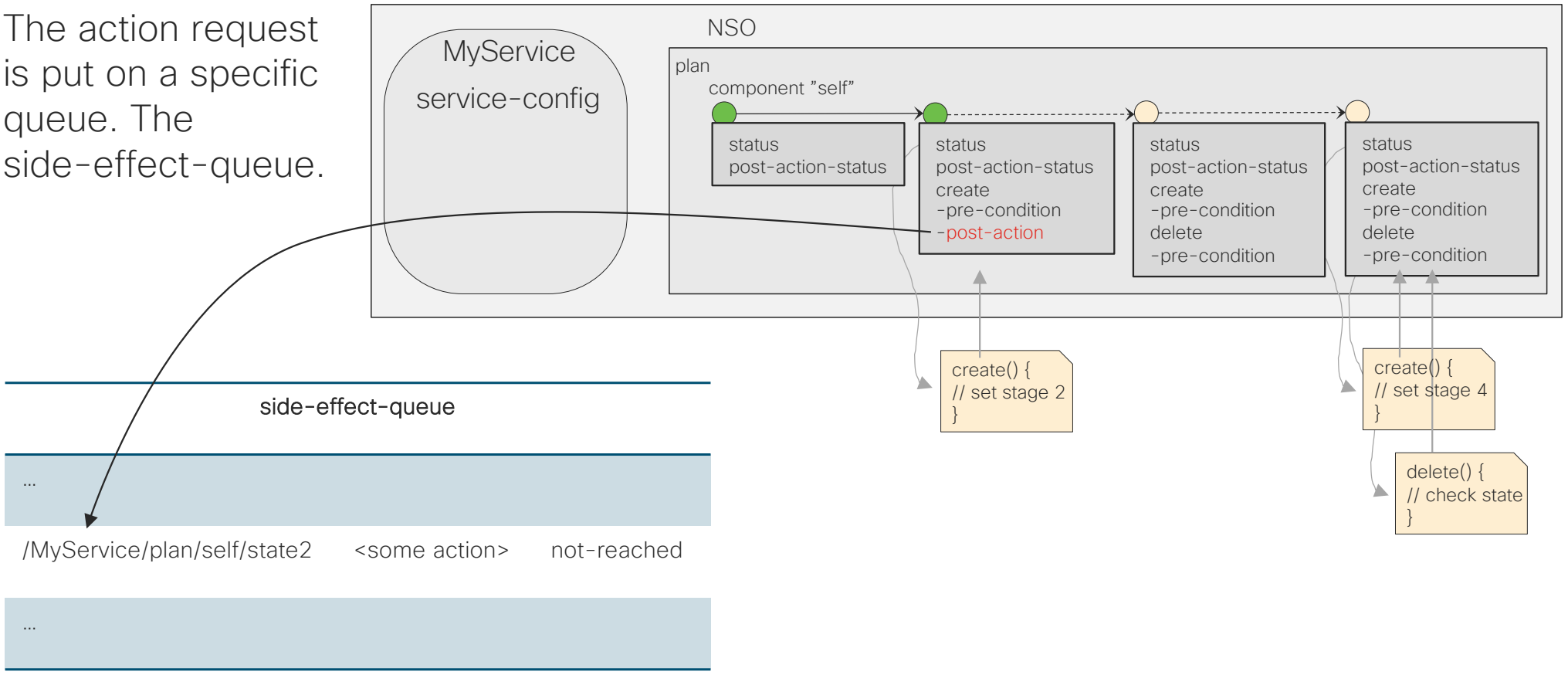
Post-actions aka. Side-effects

When a state is completed and this state has a declared post-action, this implies that this action should be run asynchroneously.



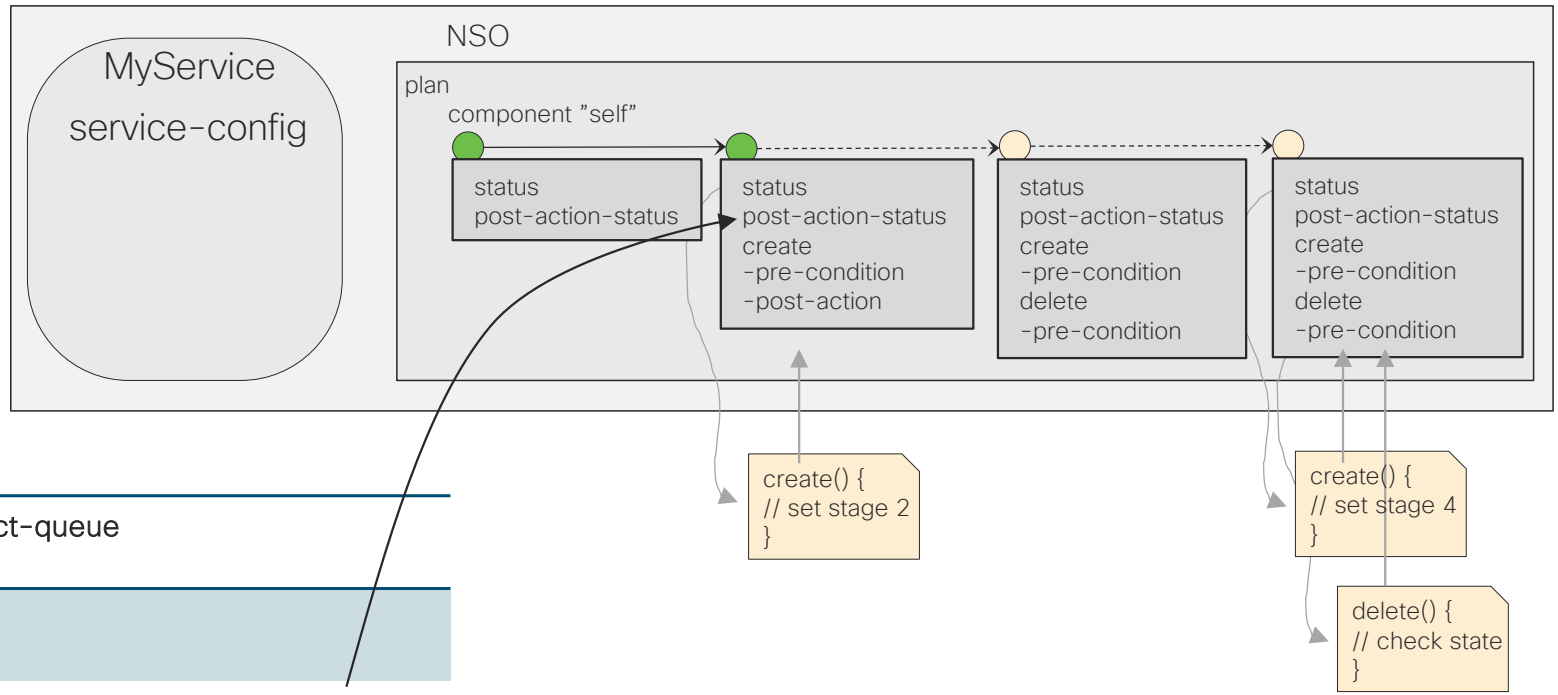
Post-actions, adding to the side-effect-queue

The action request is put on a specific queue. The side-effect-queue.



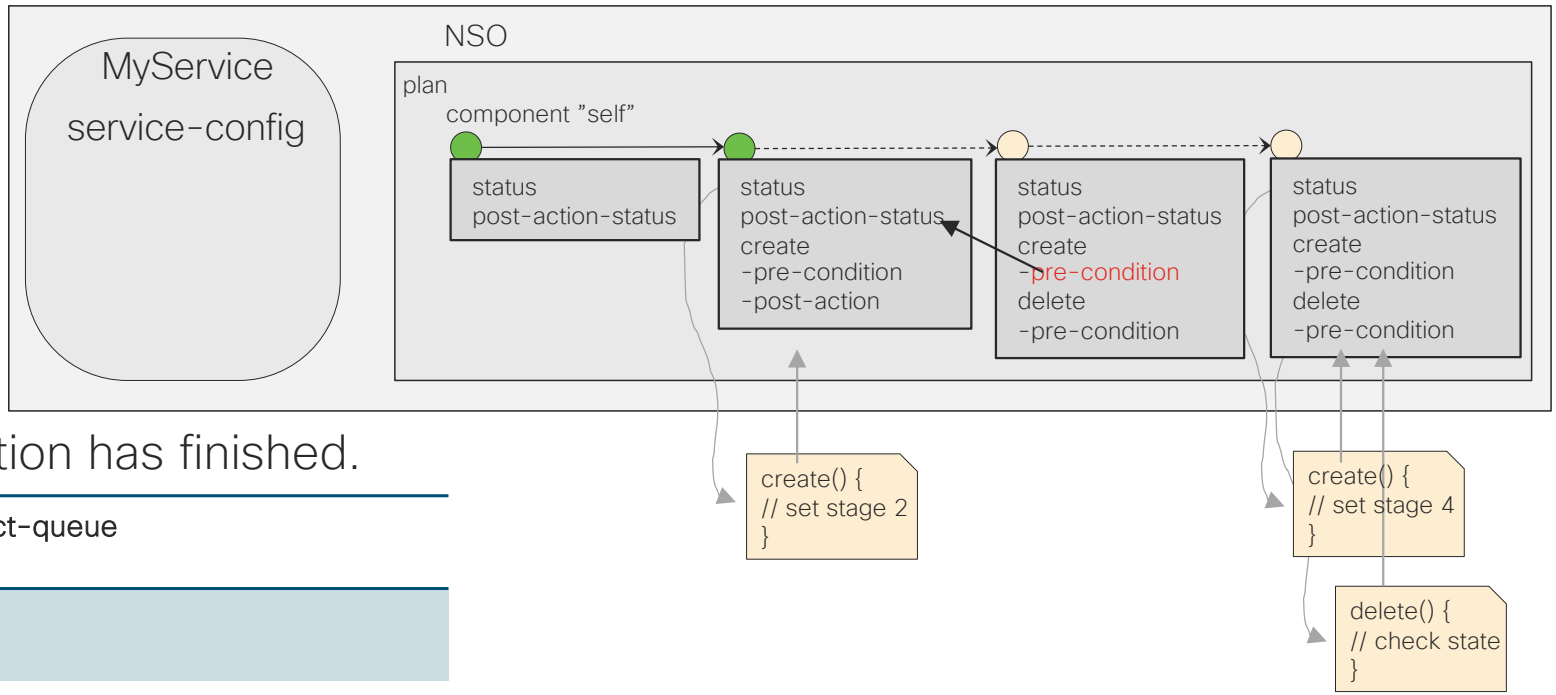
Post-actions, updating post-action-status

After execution the result is reported back to the original state.



Post-actions, waiting for completion

Next state can have a pre-condition that monitors the result of the post-action. In effect, the service will commit current transaction and postpone further execution until the action has finished.



side-effect-queue

...		
/MyService/plan/self/state2	<some action>	reached
...		

So far the plan has been static. How can we make the plan change?

During service execution sometimes decisions to change the plan is made. In traditional RFMs this is made programmatically.

A Nano service has a *behavior-tree* that makes all decisions on the layout of the plan.

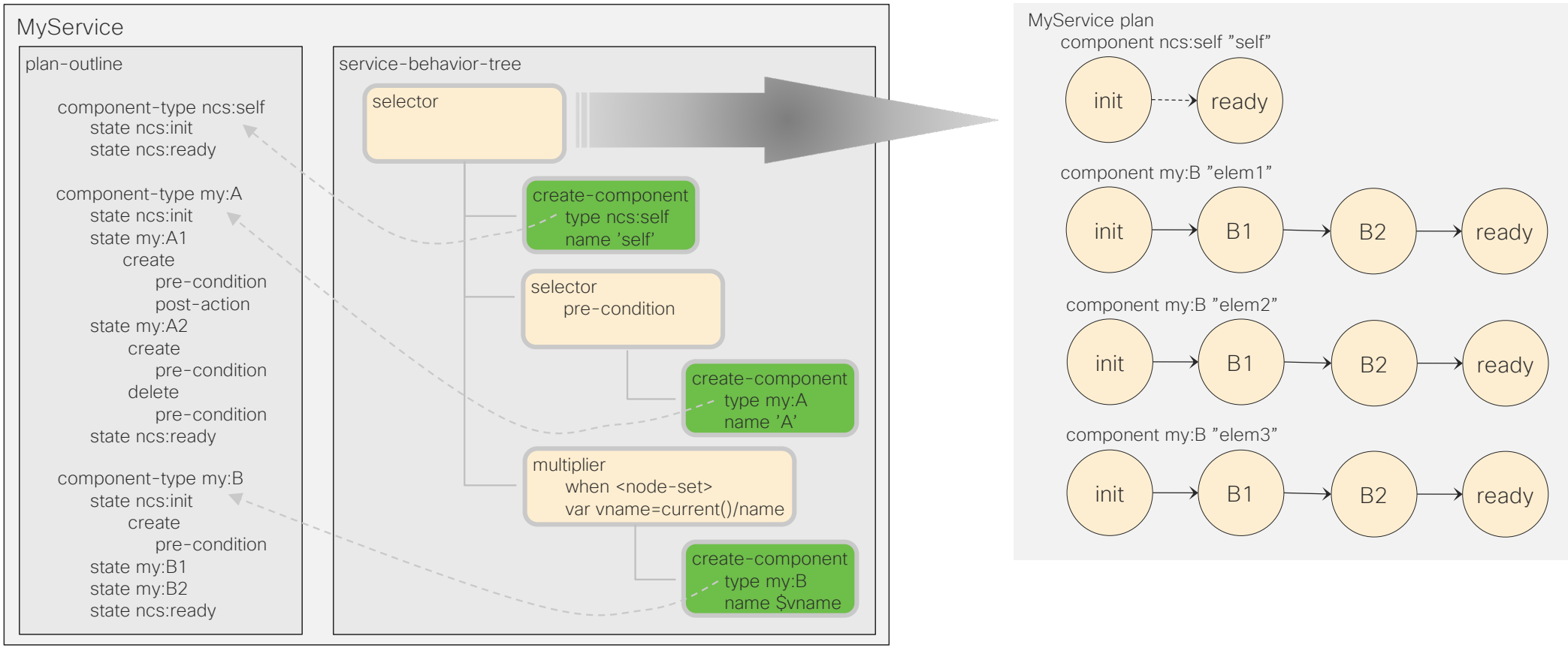
Behavior tree

- A directed tree consisting of control-flow and execution nodes with exactly one root which must be a control-flow node.
- All interior nodes are control-flow nodes and all leafs are execution nodes.
- A control-flow node can, when evaluated, decide which children should also be evaluated.
- A execution node can, when evaluated, create a plan component by referring to a component-type and give it a name.
- Evaluating the full tree renders a plan, aka. *synthesizing the plan*.

Behavior tree synthesizing cont.

- During the plan execution. The behavior tree is synthesized before the first component and then between each component that is processed.
- If a component is added this is also executed. If a component is removed it will immediately be backtracked.
- There are two types of control-nodes: selectors and multipliers. The first selects it children, the second multiplies it children depending on an controlling expression.
- Enough to create a very dynamic plan if needed.

Synthesizing the plan



How are Nano services defined

Everything around Nano services are declared using YANG extensions.

The only code part is the create code and the post actions.

Nano service declarations.

- A Nano service uses the normal service point.
- A Nano service must use the ncs:nano-plan-data grouping.
- A Nano service must declare a ncs:plan-outline with all its possible components and their states.
- A Nano service must declare a ncs:service-behavior-tree with its control-flow and execution nodes.
- All “reasonable” declarations are xpath expressions (i.e. almost everything). This includes pre-conditions, post-actions, variable declarations etc.

Nano Service declaration

```
list link {
    description "This is my Nano service";

    uses ncs:nano-plan-data;
    uses ncs:service-data;
    ncs:servicepoint link-servicepoint;

    key name;

    leaf name {
      type string;
    }

    ...

}
```

Nano Service plan-outline declaration

```
ncs:plan-outline link-plan {
ncs:component-type "ncs:self" {
      ncs:state "ncs:init";
      ncs:state "ncs:ready";
    }
    ncs:component-type "link:vlan-link" {
      ncs:state "ncs:init";
      ncs:state "link:dev-setup" {
        ncs:create {
          ncs:nano-callback;
        }
      }
      ncs:state "ncs:ready" {
        ncs:create {
          ncs:pre-condition {
            ncs:monitor "$SERVICE/endpoints" {
              ncs:trigger-expr "test-passed = 'true'";
            }
          }
        }
        ncs:delete {
          ncs:pre-condition {
            ncs:monitor "$SERVICE/plan" {
              ncs:trigger-expr "component[name != 'self'][./back-track = 'false']/state[name='ncs:ready'][./status = 'reached'] or "
              + "not(current()/component[back-track = 'false'])";
            }
          }
        }
      }
    }
  }
```

Nano Service service-behavior-tree declaration

```
ncs:service-behavior-tree link-servicepoint {
  description
    "Make before brake vlan example";

  ncs:plan-outline-ref link-plan;

  ncs:selector {
    /* create a single component of type self */
    ncs:create-component "'self'" {
      ncs:component-type-ref "ncs:self";
    }

    /* create one link component per given endpoint */
    ncs:multiplier {
      ncs:foreach "endpoints" {
        ncs:variable "LINKNAME" {
          ncs:value-expr "concat(a-device, '-', a-interface,
                                 '-', b-device, '-', b-interface)";
        }

        ncs:create-component "$LINKNAME" {
          ncs:component-type-ref "link:vlan-link";
        }
      }
    }
  }
}
```

Are the development of Nano services finished now?

On the contrary, Nano services are just recently becoming adopted in real deployments.

New functionality is discussed and planned.

New Nano services functionality could be…

- Debugging tools
- More support and control of post-actions.
- Support for notification kickers.
- Drawing tool for building/redefining Nano services.
- Support for simplifying xpath expression declarations
- …


CISCO