GENIVI®

AMM OPEN DAYS

Lifecycle Subsystem Readout

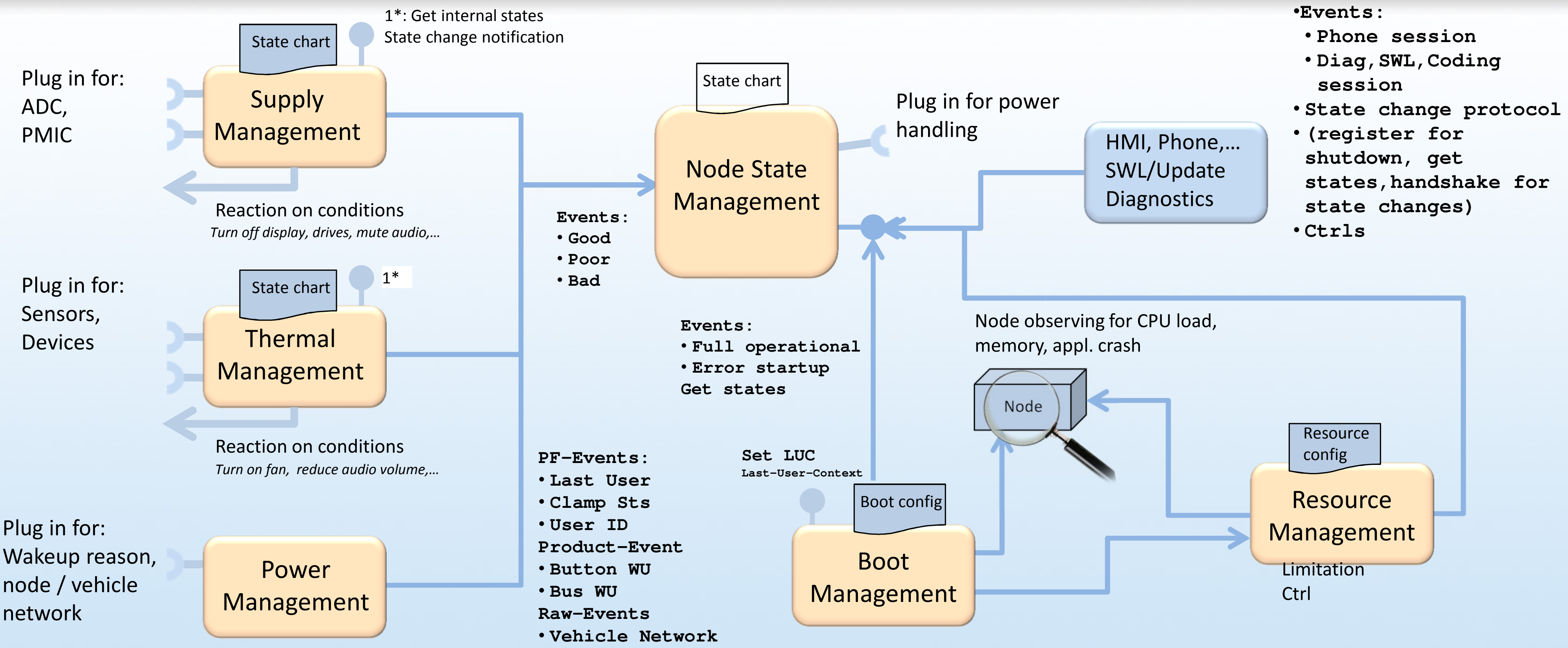
2015-10-22 09:00 | Open Projects Track

Gianpaolo Macario
GENIVI EG-SI Architect – Mentor
(slides by David Yates – Continental)

Agenda

- Introduction, goals of session
- Lifecycle Manifest
- Startup/Shutdown
- Health Management
- Lifecycle Use cases
- Open Source Status & Roadmap
- Links
- Wrap Up & Questions

Lifecycle Overview



1*: Get internal states
State change notification

Plug in for:
ADC,
PMIC

State chart

Supply Management

Reaction on conditions
Turn off display, drives, mute audio,…

Plug in for:
Sensors,
Devices

State chart

1*

Thermal Management

Reaction on conditions
Turn on fan, reduce audio volume,…

Plug in for:
Wakeup reason,
node / vehicle
network

Power Management

Events:
- Good
- Poor
- Bad

State chart

Node State Management

Plug in for power handling

Events:
- Full operational
- Error startup

Get states

PF-Events:
- Last User
- Clamp Sts
- User ID

Product-Event
- Button WU
- Bus WU

Raw-Events
- Vehicle Network

Set LUC
Last-User-Context

Boot config

Boot Management

HMI, Phone,…
SWL/Update
Diagnostics

- Events:
  - Phone session
  - Diag, SWL, Coding session
- State change protocol
- (register for shutdown, get states, handshake for state changes)
- Ctrls

Node observing for CPU load, memory, appl. crash

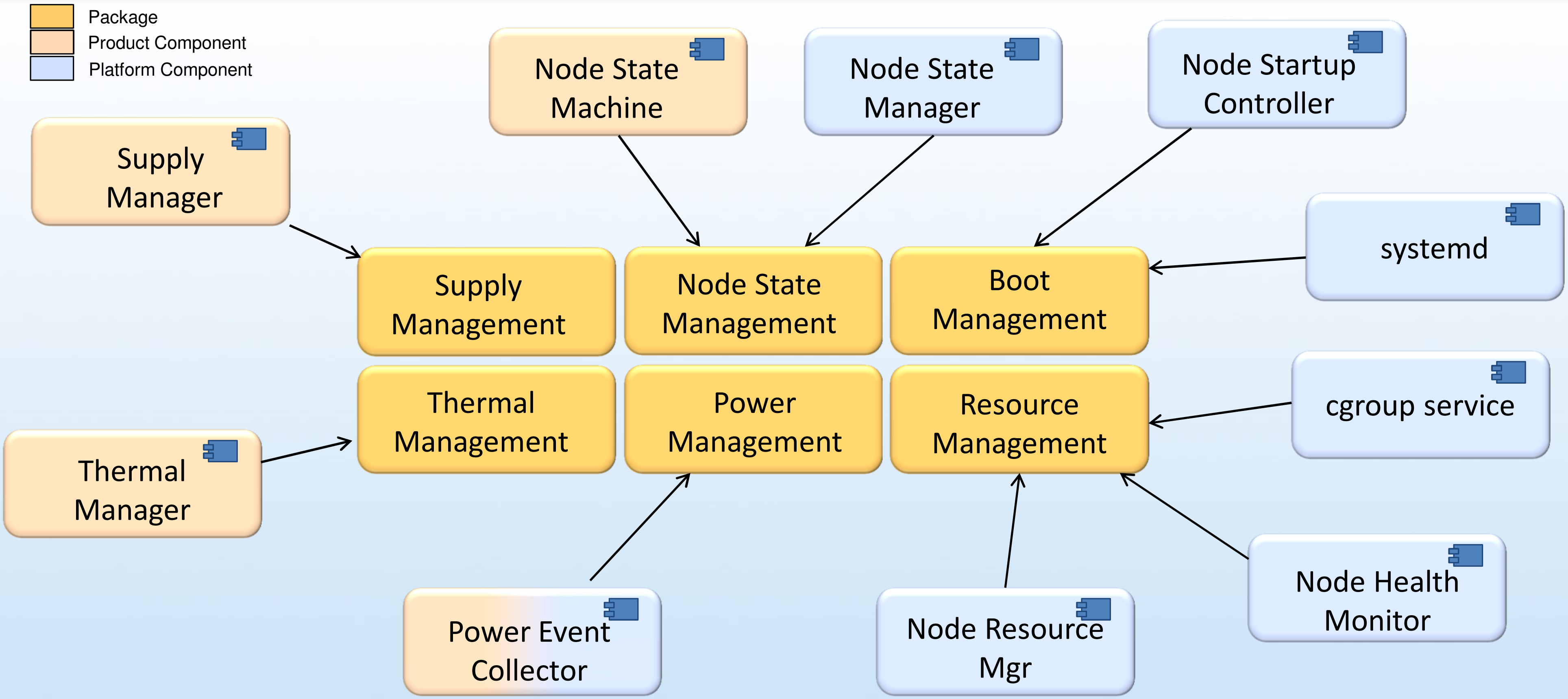
Node

Resource config

Resource Management

Limitation
Ctrl

Lifecycle Manifest



- 🟨 Package
- 🟧 Product Component
- 🟦 Platform Component

Node State Machine

Node State Manager

Node Startup Controller

Supply Manager

Supply Management

Node State Management

Boot Management

systemd

Thermal Management

Power Management

Resource Management

cgroup service

Thermal Manager

Power Event Collector

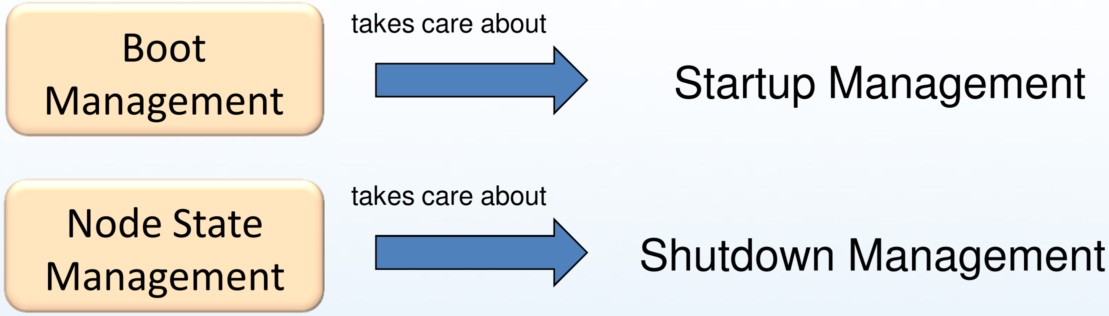
Node Resource Mgr

Node Health Monitor

Status and Roadmap

		Kronos	Leviathan	Miranda	N?
systemd Cgroup (Kernel)	Adopted component, provided by the OSS community	*specific*	*specific*	*specific*	*specific*
Node Startup Controller	GENIVI funded OSS component (implemented by Codethink)	*specific*	*specific*	*specific*	*specific*
Node State Manager	OSS Component (implemented and maintained by Continental)	*specific*	*specific*	*specific*	*specific*
Node State Machine	Product specific library	*n/a*	*n/a*	*n/a*	*n/a*
Node Resource Mgr	Implemented by Continental (OSS release upcoming)	*placeholder*	*abstract*	*abstract*	*specific*
Node Health Monitor	OSS Component (implemented and maintained by Continental)	*specific*	*specific*	*specific*	*specific*

Startup/Shutdown Management



Boot Management

takes care about

Startup Management

Node State Management

takes care about

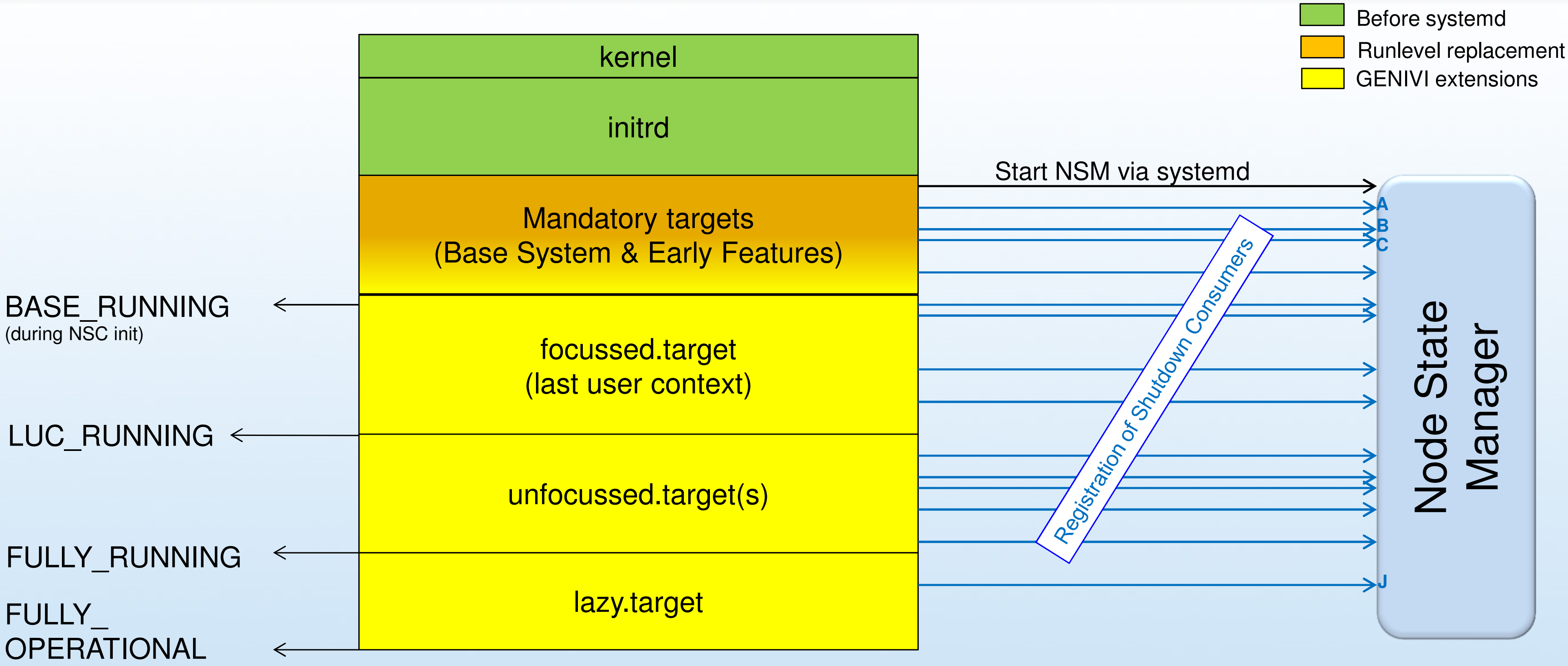
Shutdown Management

Why do we have this split?
systemd stops and unloads all components during its shutdown concept. This requires a lot of time to make them functional again in the event of a cancel shutdown.

An IVI system must be able to resume operation without losing any context and without the need for a reboot. Therefore Node State Management will only call registered consumers in the shutdown phase. This event notification will drive the components into a stable state and ensure that everything has been stored which will be needed for the next startup.

With this approach components would not be shutdown which is required for certain exceptions like the flash file system. Therefore additionally the shutdown management concept will include/use the systemd shutdown concept , where appropriate for legacy/critical components.

Shutdown preparation in Startup Phase



🟩 Before systemd
🟧 Runlevel replacement
🟨 GENIVI extensions

kernel

initrd

Mandatory targets
(Base System & Early Features)

focussed.target
(last user context)

unfocussed.target(s)

lazy.target

BASE_RUNNING
(during NSC init)

LUC_RUNNING

FULLY_RUNNING

FULLY_
OPERATIONAL

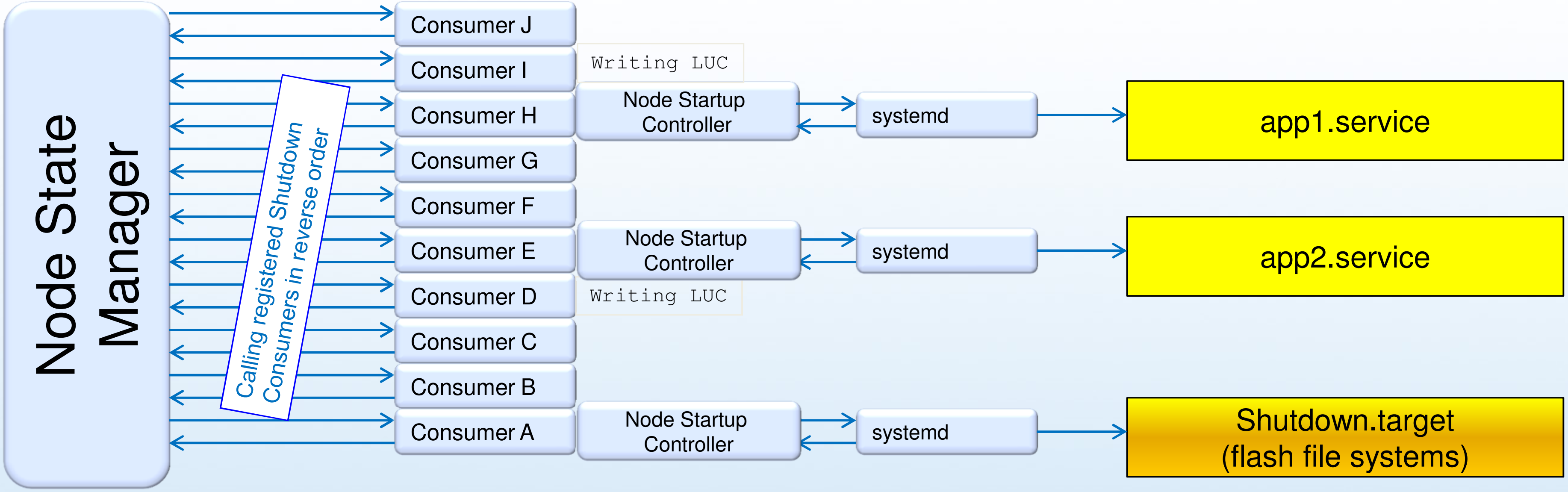
Start NSM via systemd

Registration of Shutdown Consumers

A
B
C
J

Node State Manager

Shutdown Execution



Enables:

1. Shutdown activities are trigger able without unloading the components.
2. Legacy components can be shut down in their traditional way.
3. Full flexibility on where to integrate systemd based shutdown units.

Unit Files

```
Unit]
Description=Application launcher daemon
After=mnt-appdata.mount

[Service]
Type=notify
ExecStart=/usr/bin/al-daemon --start -v
PIDFile=/tmp/al-daemon.pid

TimeoutStartSec=2
WatchdogSec=10
Restart=on-failure
StartLimitInterval=15
StartLimitBurst=5
StartLimitAction= reboot

 [Install]
WantedBy=unfocussed.target
```

What can be seen in this example is that the executable al-daemon will be run with the command options "–start –v". It will notify systemd when it has completed its initialization.

With regards to dependencies, it can only be started after mnt-appdata.mount is available and it is wanted by unfocussed.target.

This means that whenever unfocussed.target is required this unit will be included.

The application will complete initialization within 2 seconds and wants to be monitored during runtime. The application will be deemed to have failed if it has not sent a heartbeat (sd_notify) within 10 seconds. If it fails more than 5 times within a period of 15 seconds then systemd will assume a critical failure and initiate a system restart

Interfaces

The Node State Manager (NSM) will provides 2 D-Bus interfaces

- org.genivi. NodeStateManager.Consumer
- org.genivi. NodeStateManager.LifecycleControl

The “Consumer” interface is publicly available in the system and should be used by any applications that are interested in the information that the NSM maintains.

The “LifecycleControl” interface will contain functions that are secured via DBUS policies to restrict their usage. There should be a minimal subset of applications with access.

These interfaces are documented in the GENIVI UML Model (Enterprise Architect trunk) under

GENIVI Model -> Logical View -> SW Platform Components -> Node State Manager -> Interfaces

and the use cases shown in parallel today from the model can be found under

GENIVI Model -> Logical View -> Use Case Realizations -> Lifecycle

For those without access to EA you can also find exported versions of the model that are regularly updated in the wiki

System Startup

The Node Application Mode (NAM) is a Node specific data item that is used to define the functionality level (systemd runlevel) that should be achieved in the current Lifecycle.

Only one of the following example modes can be active at any one time :

- Parking
- Factory
- Transport
- SWL

The data item will be updated in the Persistence via the Node State Manager when requested through the secured method **SetApplicationMode** available under org.genivi.NodeStateManager.LifecycleControl

The data item will be read early in the startup sequence by a Lifecycle Support Library that will then define the target file to be started by systemd

To see how the update method should be used please see use case ("WB – NSM – Updating the Node Application Mode")

System Shutdown

- The **ShutdownReason** is a node-specific D-Bus property available under the org.genivi.NodeStateManager.Consumer interface

- Updated via the NSM when an event in the system triggers the shutting down of the system (i.e Thermal Management reports a dangerously high temperature)

- Consumers that want to be notified when the node will be shutdown must use the interface **Consumers.RegisterShutdownClient** and must provide their own method called **LifecycleRequest** that will be called by the NSM in the event of a shutdown (or cancel shutdown)
- Additionally they must specify the type of shutdown they are registering for (i.e fast or Normal as defined in **NSM_ShutdownTypes_e)**

- For example sequence diagrams in this area please see
“WB – NSM – Application Blocking system shutdown” and
“WB – NSM – Cancel shutdown ok”

Session Management

- The Node Session State (NSS) contains information about the current sessions that are active in that Head Unit and are used by the NSM state machine to determine correct actions to events

- The state of a current session can be read using the method **Consumers.GetSessionState** with the name of the session and can be set using **Consumers.SetSessionState** ("WB – NSM – Application Blocking system shutdown")

- Alternatively consumers can register to be signalled ("**SessionStateChanged**") when a particular Session State has changed ("WB – NSM – Session State Handling")

- To control new sessions the methods **Consumers.RegisterSession** and **UnRegisterSession** have been provided ("WB – NSM – Add new session state")

Node State

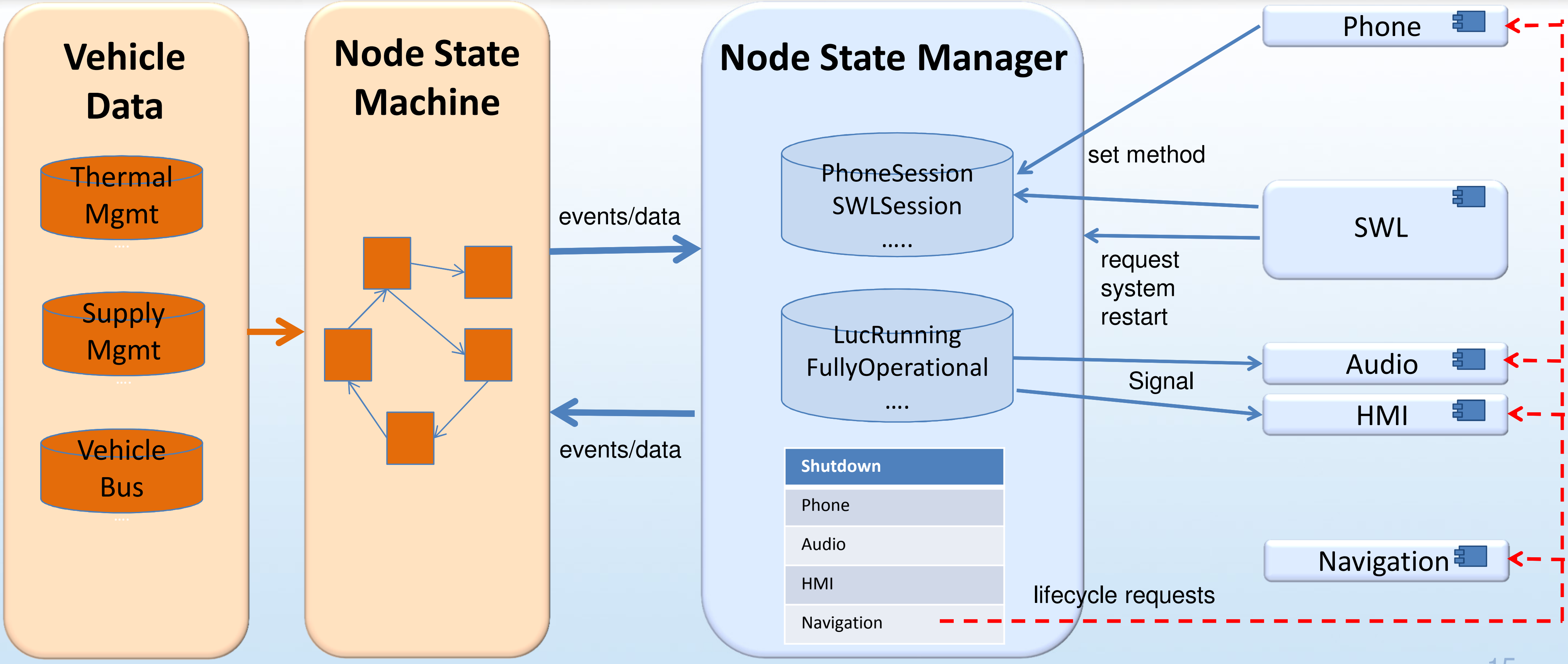
- The Node State is a node-specific data item used to track the startup state of the Head Unit.
- It is updated via the Node State Manager based on internal state changes or when triggered by an application through the **LifecycleControl.SetNodeState** method.

- Mandatory platform states are defined in the enumeration **NSM_NodeState_e** but there will not be range checking on the interface so product states can be added

- The current node state can be accessed by using the method **Consumers.GetNodeState** or consumers can register to be signalled when the Node State is updated. This signal is sent to registered clients and will include the current Node State as a parameter

- For a use case showing the Node State please see use case "WB – NSM – Application Blocking system shutdown"

Use Cases



Vehicle Data

Thermal Mgmt

Supply Mgmt

Vehicle Bus

Node State Machine

events/data

events/data

Node State Manager

PhoneSession
SWLSession
.....

LucRunning
FullyOperational
....

Shutdown
Phone
Audio
HMI
Navigation

set method

request
system
restart

Signal

lifecycle requests

Phone

SWL

Audio

HMI

Navigation

Resource Management - Goals

- Resource management contains the functionality to ensure that the node runs in a stable and defined manner.

- To do this, it will monitor and limit different aspects of SW component behavior including system resources (i.e. CPU load and memory) and critical run-time observation.

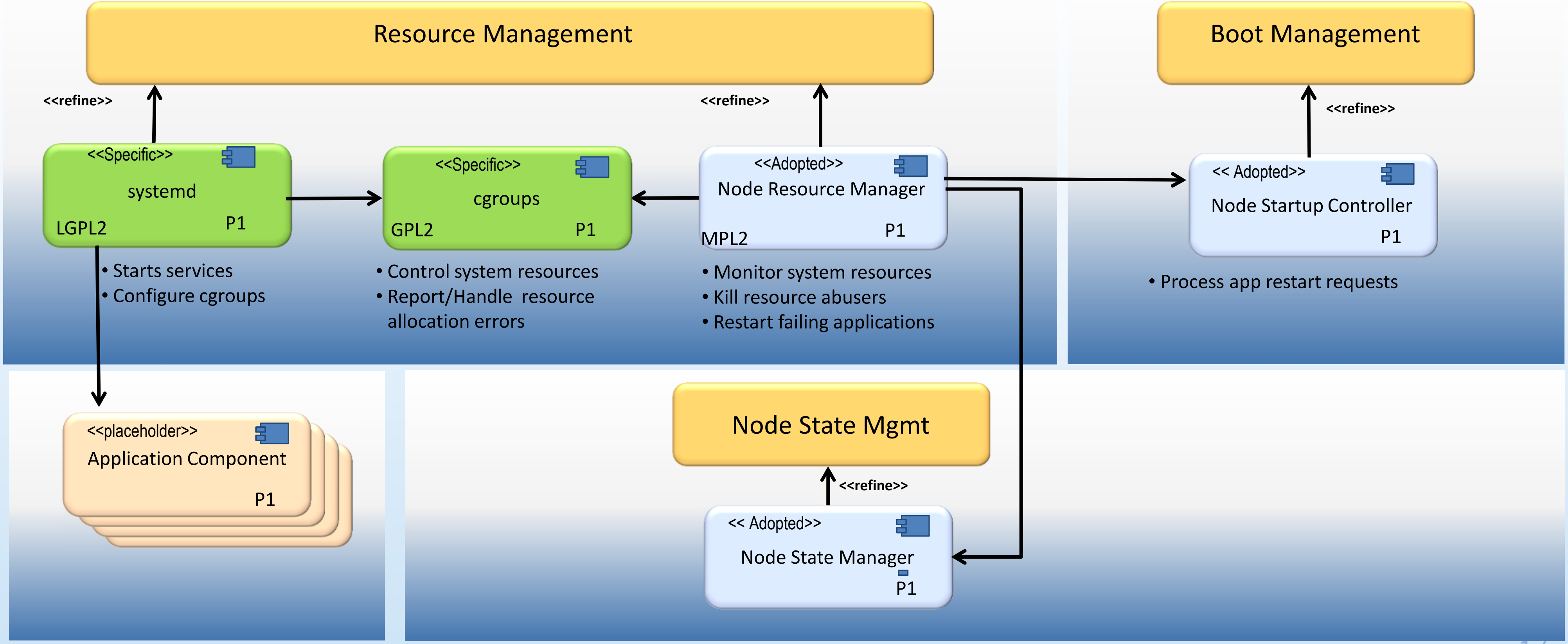
- Resource allocation will be configurable on a component basis through the use of cgroups.

- Health Management provides a configurable escalation strategy defining actions to be taken in the case of system failures.

- For instance, in the case of run-time observation failure, the following escalation strategies could be considered to repair the situation :
  - Restart the application
  - Restart the node
  - Rollback of application to previous version
  - Rollback of all user updates to baseline state
  - Deletion of applications persistence data
  - Deletion of all user persistence data

- What is not included is security handling for resources (i.e. restricted access to resources)!!!

Resource Management



- Starts services
- Configure cgroups

- Control system resources
- Report/Handle resource allocation errors

- Monitor system resources
- Kill resource abusers
- Restart failing applications

- Process app restart requests

Overview of cgroups

cgroups (control groups) is a Linux kernel feature which can limit, account and isolate resource usage (CPU, memory, disk I/O, etc.) of process groups.

The following are examples of cgroup handling:

- **Resource limiting:** groups can be set to not exceed a set memory limit
- **Prioritization:** some groups may be configured to get a larger share of CPU or disk I/O throughput
- **Accounting:** to measure how many resources certain systems use for e.g. system debugging and performance analysis/tuning
- **Isolation:** separate namespaces for groups, so they don't see each other's processes, network connections or files
- **Control:** freezing groups or check pointing and restarting

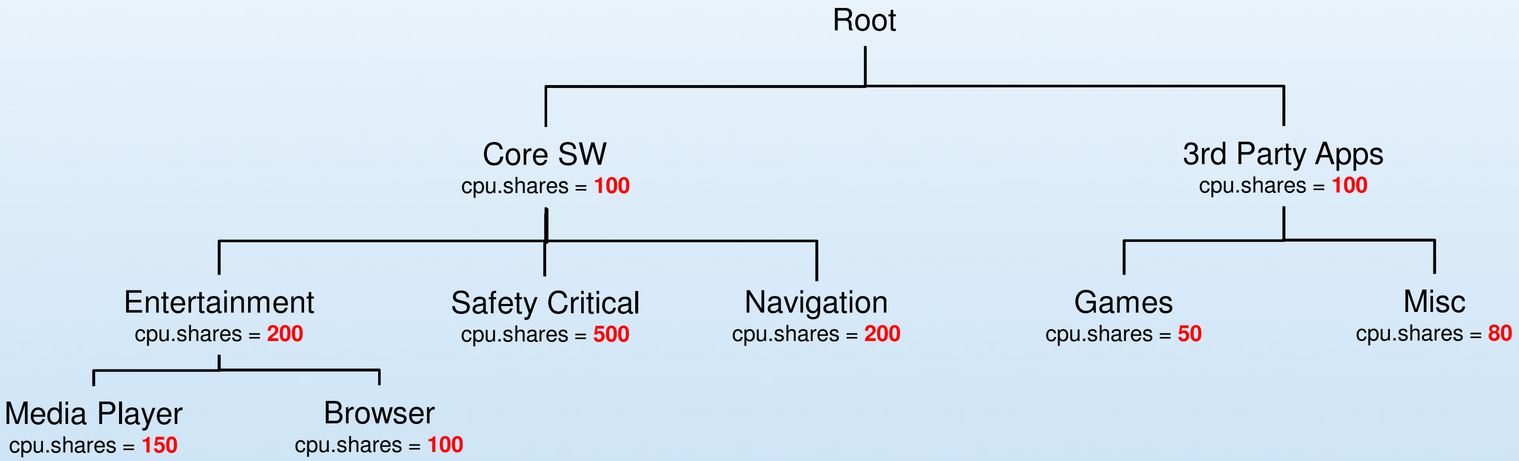
A control group is a collection of processes that are bound by the same criteria. These groups can be hierarchical, where each group inherits limits from its parent group.

Tree Hierarchies

The tree below shows an example of how a set of standard Automotive SW could be split. We have 1 hierarchy with the CPU subsystem attached that contains 10 cgroups . Each cgroup has been configured with specific subsystem values.

Now when we allocate processes to certain groups we will be able to control the CPU used by those processes. We will also be able to guarantee that our “Safety Critical” processes will always have access to ~33% of the CPU.

NOTE: These values do not specify a max. amount of CPU that a group can use rather they specify the amount of CPU they are guaranteed if they need it. Also this is a simple example and may not match the exact definition provided within recent systemd versions



Integration with systemd

systemd supports the creation, configuration and population of cgroups during the start phase through the use of the standard systemd unit files.

Using the unit files it is possible to specify the

- “ControlGroup=“ – define the control groups the new processes shall be members of
  - Takes a space-separated list of cgroup identifiers. A cgroup identifier has a format like cpu:/foo/bar, where "cpu" identifies the kernel control group controller used, and /foo/bar is the control group path
- “ControlGroupModify=“ - Takes a boolean argument. If true, the control groups created for this unit will be owned by the user specified with ”User=“ (and the appropriate group), and he/she can create subgroups as well as add processes to the group
  - It is suggested that this argument is left to false to ensure control of the system
- “ControlGroupPersistent=“ - Takes a boolean argument. If true, the control groups created for this unit will be marked to be persistent, i.e. systemd will not remove them when stopping the unit. The default is false, meaning that the control groups will be removed when the unit is stopped

Integration with systemd (cont.)

- “ControlGroupAttribute=“ - Sets a specific control group attribute for executed processes, and (if needed) add the executed processes to a cgroup in the hierarchy of the controller the attribute belongs to.
  - Takes two space-separated arguments: the attribute name (syntax is cpu.shares where cpu refers to a specific controller and shares to the attribute name), and the attribute value.
  - Example: ControlGroupAttribute=cpu.shares 512. If the attribute belongs to a kernel controller hierarchy the unit is not already configured to be added to (for example via the “ControlGroup=“ option) then the unit will be added to the controller and the default unit cgroup path is implied.
  - Thus, using “ControlGroupAttribute=“ is in most case sufficient to make use of control group enforcements, explicit ”ControlGroup=“ are only necessary in case the implied default control group path for a service is not desirable.

For more details about control group attributes see cgroups.txt. This option may appear more than once, in order to set multiple control group attributes.

Unit Files extended

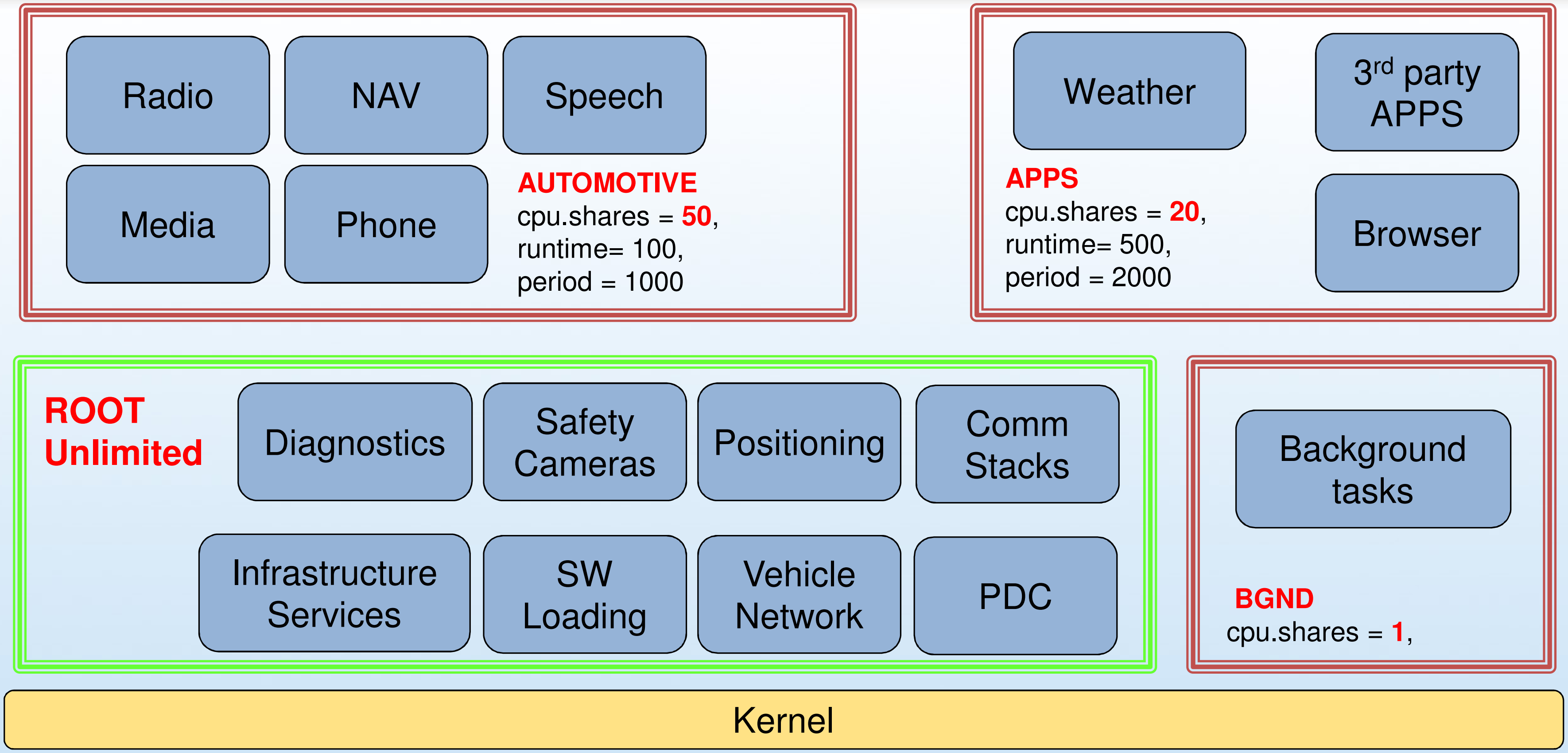
```
[Unit]
Description=Application launcher daemon
After=mnt-appdata.mount

[Service]
Type=forking
RemainAfterExit=yes
ExecStart=/usr/bin/al-daemon --start -v
TimeoutStartSec=2
WatchdogSec=10
Restart=on-failure
StartLimitInterval=15
StartLimitBurst=5
StartLimitAction= reboot
ControlGroup=apps
CPUShares=512
MemoryLimit=2M
MemorySoftLimit=1M

 [Install]
WantedBy=unfocussed.target
```

Now you can see that cgroup information has been added to state that this application will be part of the “apps” cgroup and that the group if not already defined at this point will have 512 shares of the CPU and will have access to 2M of memory.

Additionally the cgroup “apps” will have a soft limit of 1MB. This means that if the group goes over the limit then the Node Resource Manager will take actions.

At the minute it is planned that a high layer application will be notified of this soft limit breach and he will be responsible for the action (i.e informing user about possibility to close 1 or more applications).

Node Configuration (CPU)



Radio

NAV

Speech

Media

Phone

AUTOMOTIVE
cpu.shares = 50,
runtime= 100,
period = 1000

Weather

3rd party APPS

APPS
cpu.shares = 20,
runtime= 500,
period = 2000

Browser

ROOT
Unlimited

Diagnostics

Safety Cameras

Positioning

Comm Stacks

Infrastructure Services

SW Loading

Vehicle Network

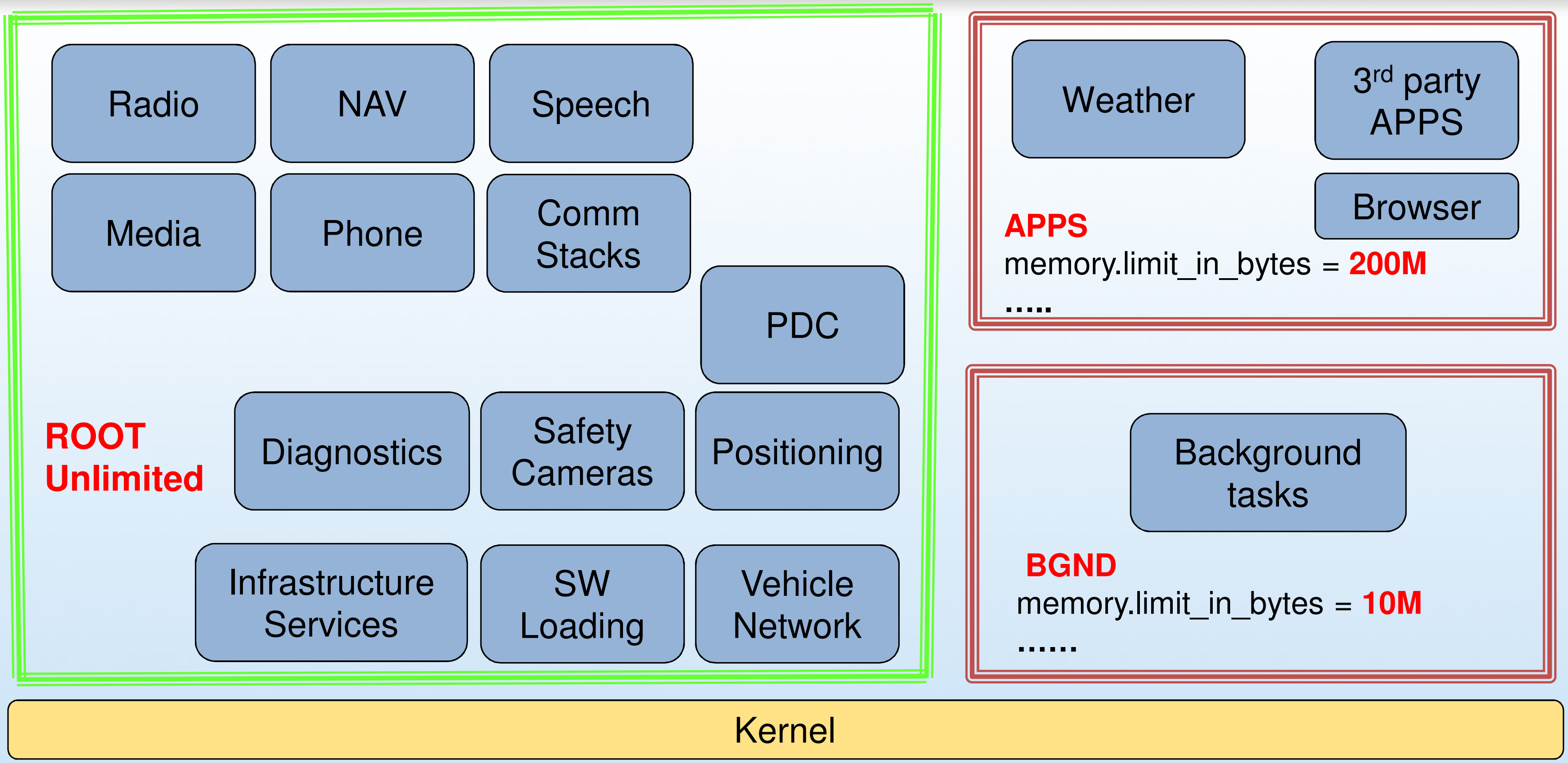
PDC

Background tasks

BGND
cpu.shares = 1,

Kernel

Node Configuration (Memory)



Radio

NAV

Speech

Media

Phone

Comm Stacks

PDC

ROOT Unlimited

Diagnostics

Safety Cameras

Positioning

Infrastructure Services

SW Loading

Vehicle Network

Weather

3rd party APPS

Browser

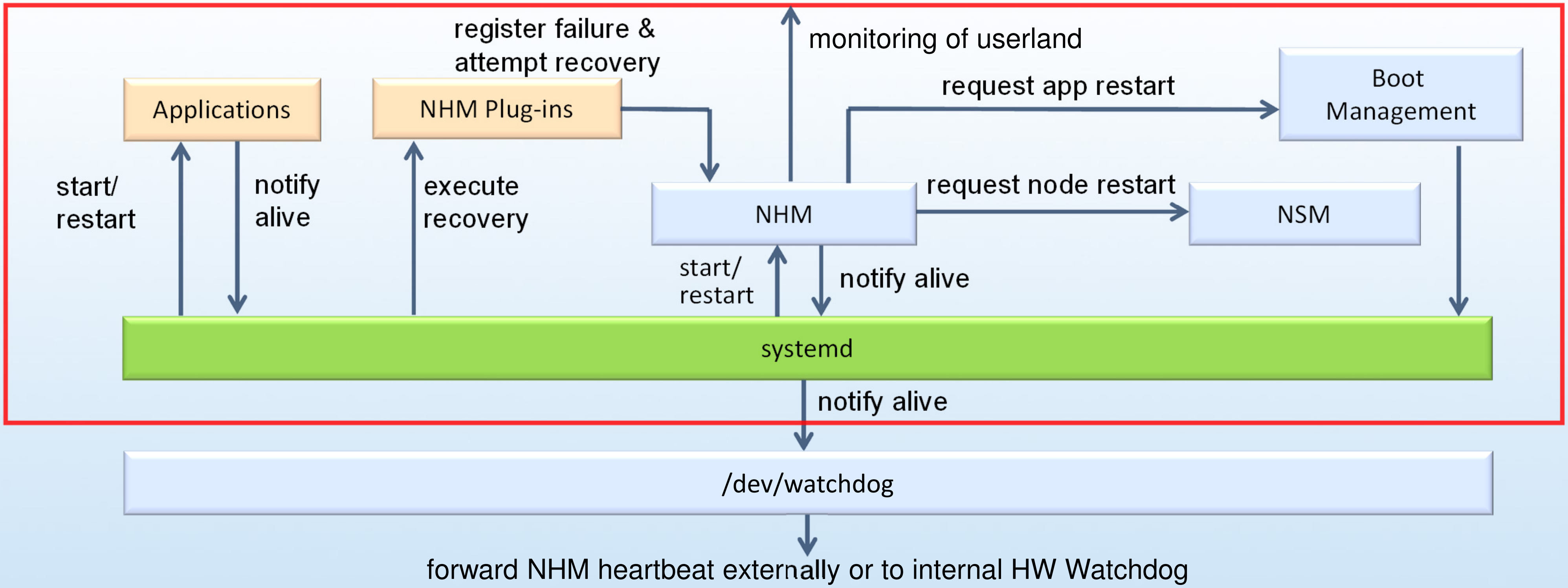
APPS
memory.limit_in_bytes = **200M**
.....

Background tasks

BGND
memory.limit_in_bytes = **10M**
......

Kernel

Health Management



forward NHM heartbeat externally or to internal HW Watchdog

Node Health Management

The Node Health Monitor will be started by systemd and will interact with application plug-ins to inform it that a component has failed in the system. He will be responsible for :

- providing an interface with which plug-ins can register failures
  - name of the failing service will be used to identify and track failures
- tracking failure statistics over multiple lifecycles for the system and components
  - the name of the failing service will be used to identify and track component failures
  - lifecycles without failures will result in a positive error count being decremented
  - statistics on number of failures in number of lifecycles will be maintained (i.e. 3 failures in last 32 lifecycles)
  - reading the wakeup, startup and shutdown reason and updating error counts accordingly to catch unexpected system restarts
- provide an interface for plug-ins to read system and component error counts
  - name of the failing service will be used to identify and track failures
- provide an interface for plug-ins to request a node restart

Node Health Management (cont.)

Additionally the Node Health Monitor will test a number of product defined criteria with the aim to ensure that userland is stable and functional. For instance it will be able to validate that :

- there is enough free system memory
- the CPU is not reporting an excessively high load for a sustained period
- defined file accessibílity is possible
- defined processes are still running
- communication is possible (DBUS)
- a user defined process can be executed with an expected result

If the NHM believes that there is an issue with user land then it will initiate a system restart by killing systemd.

Health Management – HW watchdog

It is proposed to use, when supported, a low level HW watchdog which will validate that systemd is running correctly.

The watchdog implementation will be able to initiate a complete shutdown process when it believes that a failure has occurred :

- idle init, so nothing new can be started
- kill all processes
- write a reboot record to wtmp
- turn off accounting
- turn off quota
- turn off swap
- unmount all mounted partitions

NOTE: In this scenario a normal system shutdown will not be completed therefore cached persistent data from that Lifecycle will be lost

Health Management - systemd

It is proposed to use the built in service monitoring functionality (watchdogs) that systemd provides for monitoring services and restarting them automatically on failure.

Within a service unit file it is possible to configure systemd that it will expect a heartbeat from the service within a particular time interval (WatchdogSec=). If this heartbeat is not received then systemd will decide whether it should automatically restart the service (Restart=).

To ensure that we do not get into a cyclic restart scenario it is also possible to define how often this restart action should occur (StartLimitInterval=, StartLimitBurst=). If we exceed our retry attempts then it can be configured that a failure service is started (StartLimitAction=none, OnFailure=).

This failure service is the defined Node Health Monitor plug-in for that particular service.

systemd will be monitored by the HW watchdog.

Health Management - Plugins

A Node Health Monitor plug-in will be executed by systemd when a service fails (either during startup or at runtime). It should contain enough functionality to :

- register with the Node Health Manager (NHM)
    - providing the name of the service file failing
- request the error status count from the NHM
    - providing the name of the service file failing
- based on the error count attempt recovery, for instance:
    - if a file system fails to mount then the recovery action could be to format the file system and request a node restart
    - if it is an application that has failed multiple times then we may want to delete that applications persistency data and restart the application
    - when possible, request that the SW is uninstalled or rolled back (this will require further work to identify the interaction with the SW Management team for how we can identify the package name)
- request application restart with Node Startup Controller
- request node restart via NHM

Links

Lifecycle cluster overview:

- http://wiki.projects.genivi.org/index.php/Lifecycle_cluster

Link to project pages:

- http://projects.genivi.org/node-health-monitor/
- http://projects.genivi.org/node-startup-controller/
- http://projects.genivi.org/node-state-manager/

Links to git repositories:

- http://git.projects.genivi.org/?p=lifecycle/node-health-monitor.git;a=summary
- http://git.projects.genivi.org/?p=lifecycle/node-startup-controller.git;a=summary
- http://git.projects.genivi.org/?p=lifecycle/node-state-manager.git;a=summary

Questions

Thanks for your time and attention.

Any questions??