Robotic Control System

Low Level Subsystems and Infrastructure

Low Level infrastructure

- Communications infra-structure
  - CIL
  - JMS
- Primary component sub-systems
  - ICM
  - TCM
  - EMS
  - (ISS)
- Utilities
  - Astrometry
  - Logging
  - Configuration

CIL

- Communications Interface Layer
- TTL's protocol used to communicate between various nodes in system (AMN, MCP, AZM...)
- Transport - uses UDP.
- Various headers and routing information.
- Each node has a single port assigned.
- Can only use this port – configured somewhere in the system (MCC, SCC ?).
- Used between RCS and TCS (via Network interface) – something that receives CIL packets containing TCS commands and forwards them to the TCS.

RCS TO TCS

- Outgoing CIL packets contain TCS commands.
- RCS-TCS: send command packet.
- TCS-RCS: send actioned packet on receipt.
- TCS-RCS: send completed or error packet when TCS is deemed to have completed.
- It is *believed* that the Network layer monitors the TCS/SDB to see if relevant status entries support the success of actions expected from performing the command have occurred:-
  - one reason why seemingly simple commands sometimes take a while to complete – the Network layer is waiting for its configured timeout to see if the system is in the expected state.
- Sometimes strange results occur and have to be handled by the TMS.

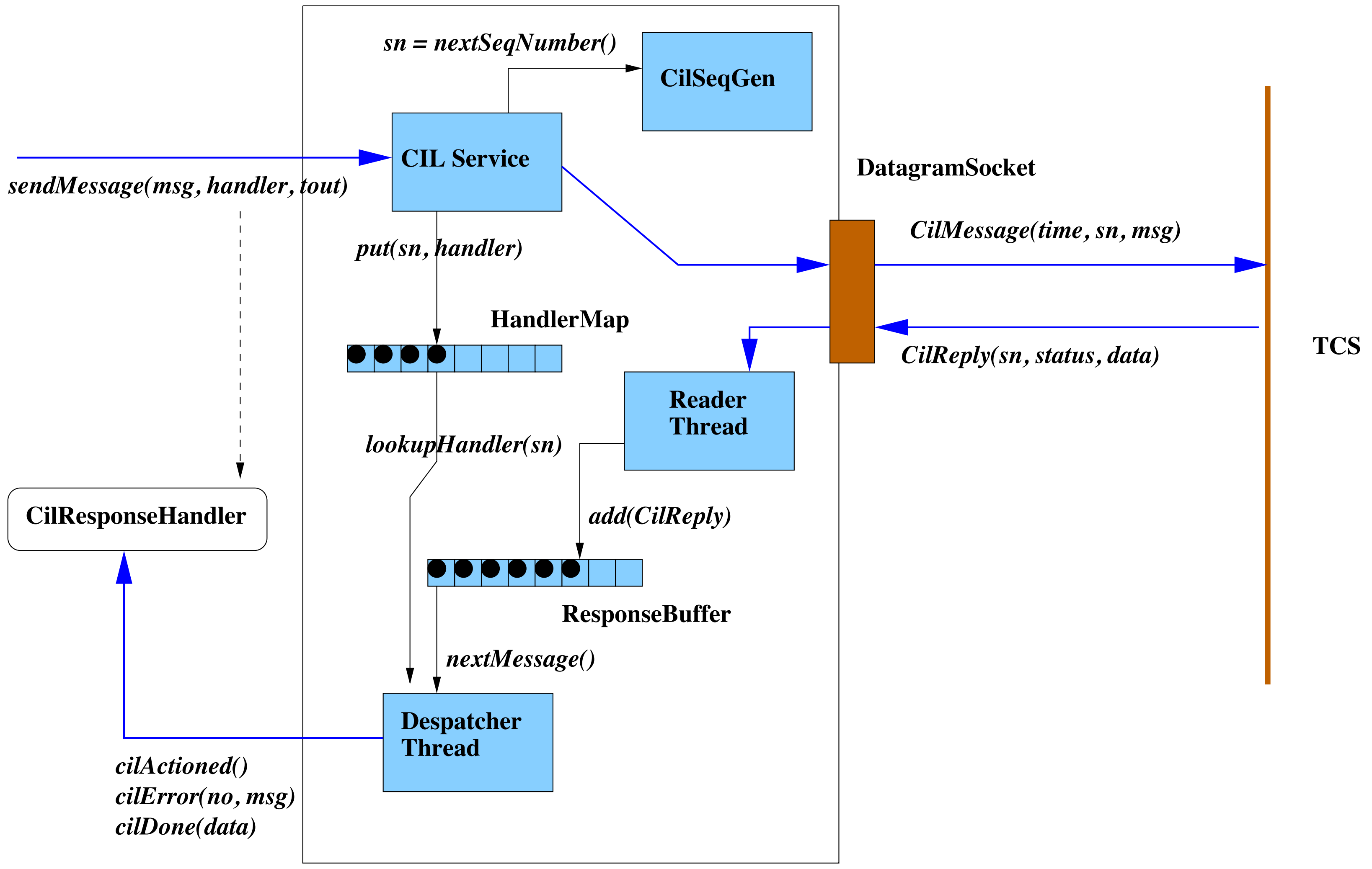
CIL SERVICE

- CIL is accessed by the RCS through a rmi based service provider.
  - *ngat.net.cil.CilService (ltdevsrv: ~dev/src/cil/)*
- All comms go through a single datagram port – incoming and outgoing.
- There is some internal buffering but it is not controllable from java layer.
- Incoming messages have to be read quickly or they can be lost.
  - RCS sends commands to control the telescope.
  - TCM sends commands to monitor telescope status.
  - EMS sends commands to monitor weather.
- These can all be happening at the same time with intermingled replies.

CIL IMPLEMENTATION

- Incoming messages are assigned a unique sequence number and mapped against a supplied handler to accept a completion callback.
- Message is packed into UDP packet along with timestamp, sequence number and some headers, then sent out to TCS (network interface).
- Returns are read by a ReaderThread and placed into a queue.
- DespatcherThread reads from queue matches sequence number to handler and makes either an *actioned*, *completed* or *error* callback

CIL SERVICE


sn = nextSeqNumber()
CilSeqGen
CIL Service
sendMessage(msg, handler, tout)
DatagramSocket
CilMessage(time, sn, msg)
put(sn, handler)
HandlerMap
CilReply(sn, status, data)
TCS
Reader Thread
lookupHandler(sn)
CilResponseHandler
add(CilReply)
ResponseBuffer
nextMessage()
Despatcher Thread
cilActioned()
cilError(no, msg)
cilDone(data)

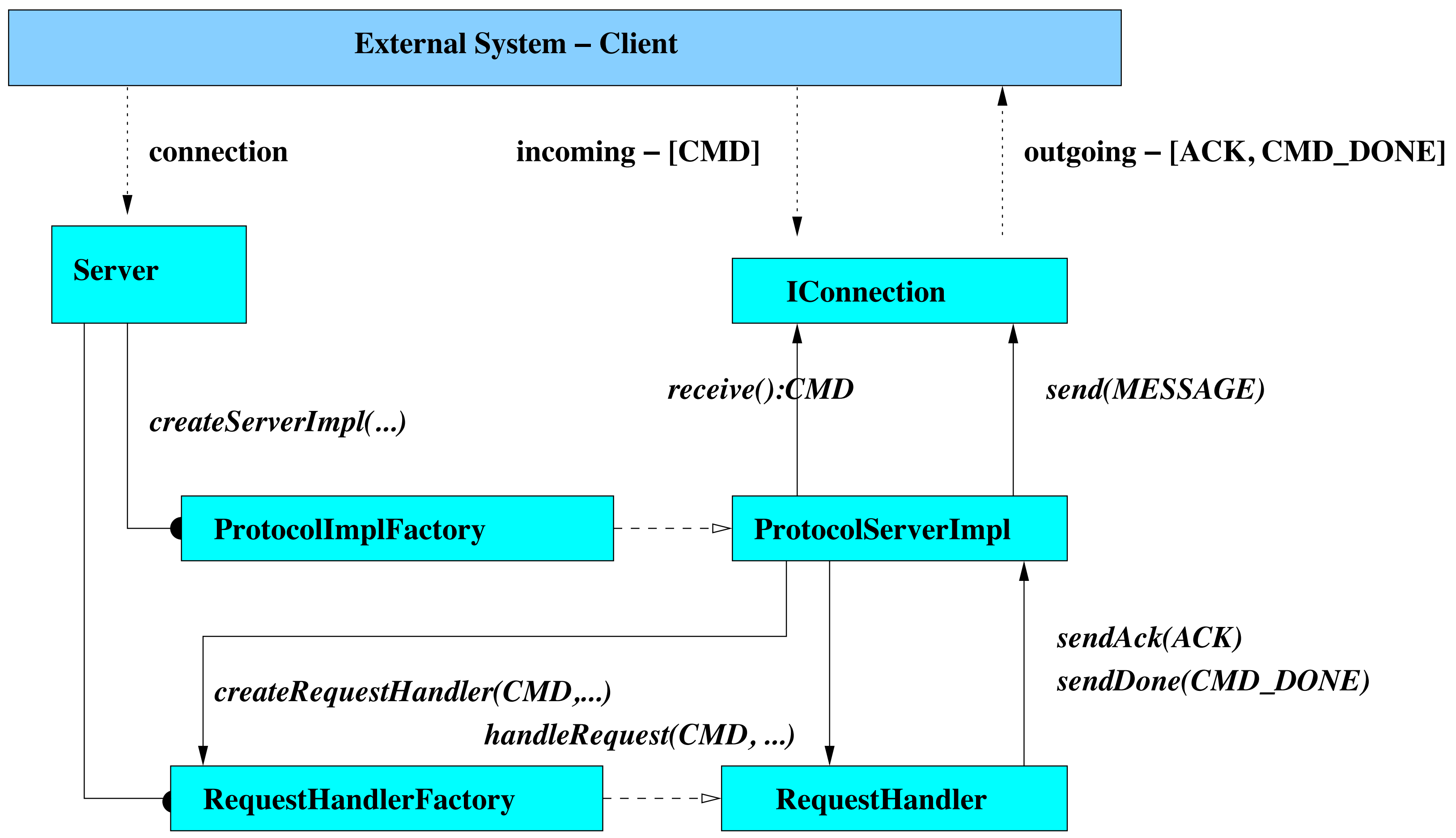
JMS

- Java Message System – passes java serialized objects over ObjectStreams between client and server systems.
- Communications between high level robotic systems.
  - RCS, ICS (RatCam, IO:O, RISE, FRODO,etc), SMS, BSS, ISS, POS. Also used by RcsGUI, IcsGUI.
- Protocol
  - Client-Server: COMMAND.
  - Server-Client: ACK (with timeout) 1 or more.
  - Server-Client: DONE (with success/data/error).
- Detailed documentation: *http://ltdevsrv/~dev/*

JMS CLASSES

- Server (ngat.net.Server) and subclasses…
  - TOC_Server, ISS_Server, POS_Server, Ctrl_Server
- Provides socket or similar endpoint
- Creates *IConnection* when client connects.
- Uses *ProtocolImplFactory* to create an instance of *ProtocolServerImpl*.
    - JMS is just one protocol supported by this architecture.
    - Also CAMP, POS and TOCS.
- *ProtocolServerImpl* reads command
- Uses *RequestHandlerFactory* to create appropriate *RequestHandler* for the specified command.
- *RequestHandler* processes command and sends Acks and Done messages to client, e.g.
  - ISS: *GET_FITSImpl* processes a *GET_FITS* command.
  - BSS: *BEAMSTEERImpl* process *BEAM_STEER* command.

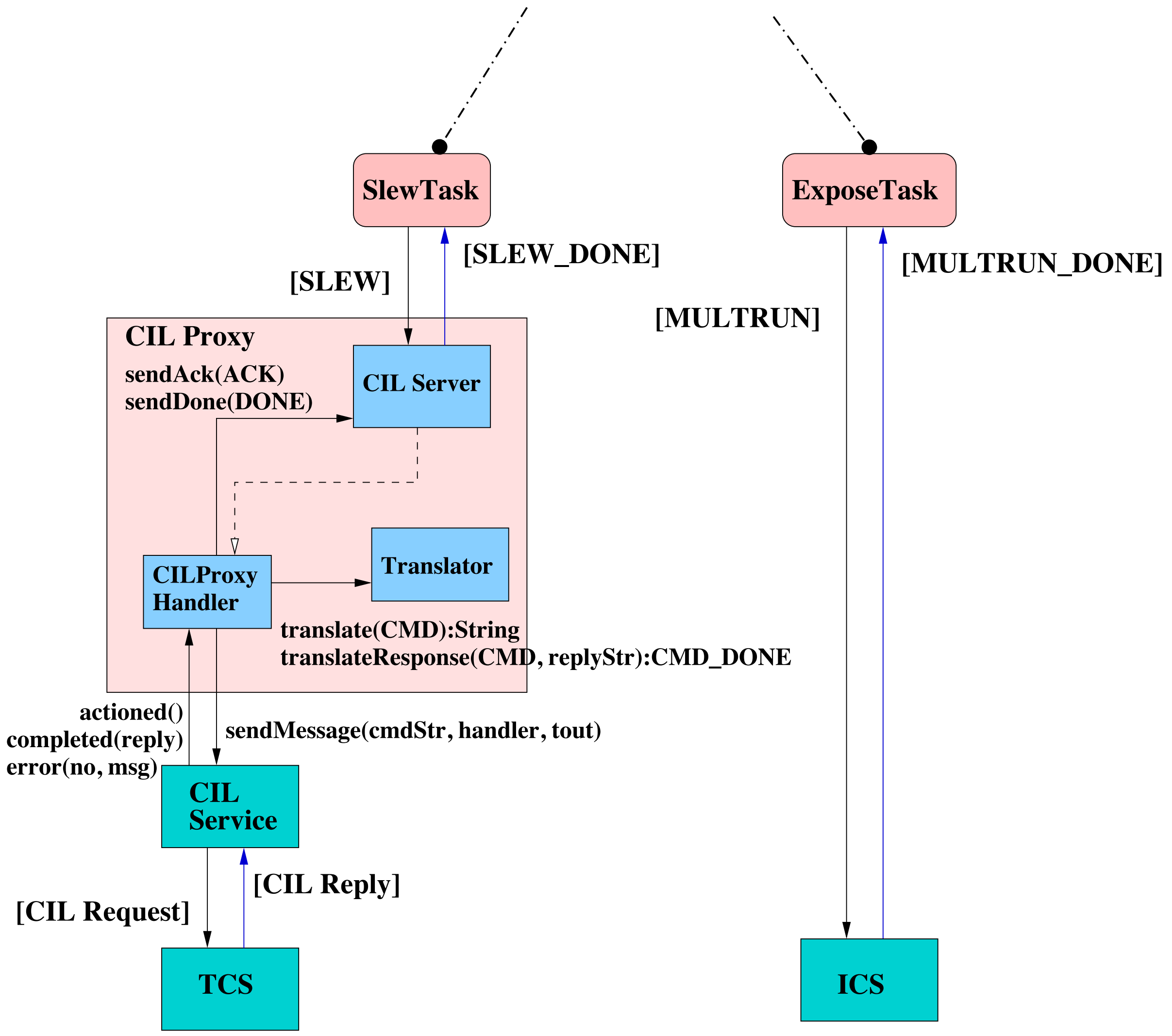
JMS Implementation



CIL MEETS JMS

- Within TMS, the executive tasks communicate with external systems via JMS.
- The TCS is accessed via CIL which is not a TCP based protocol.
- CIL_Proxy layer, *ngat.rcs.comms* package, is used to emulate a TCP like service.
- From the point of view of an executive task the TCS looks the same as any instrument other than having different commands available.

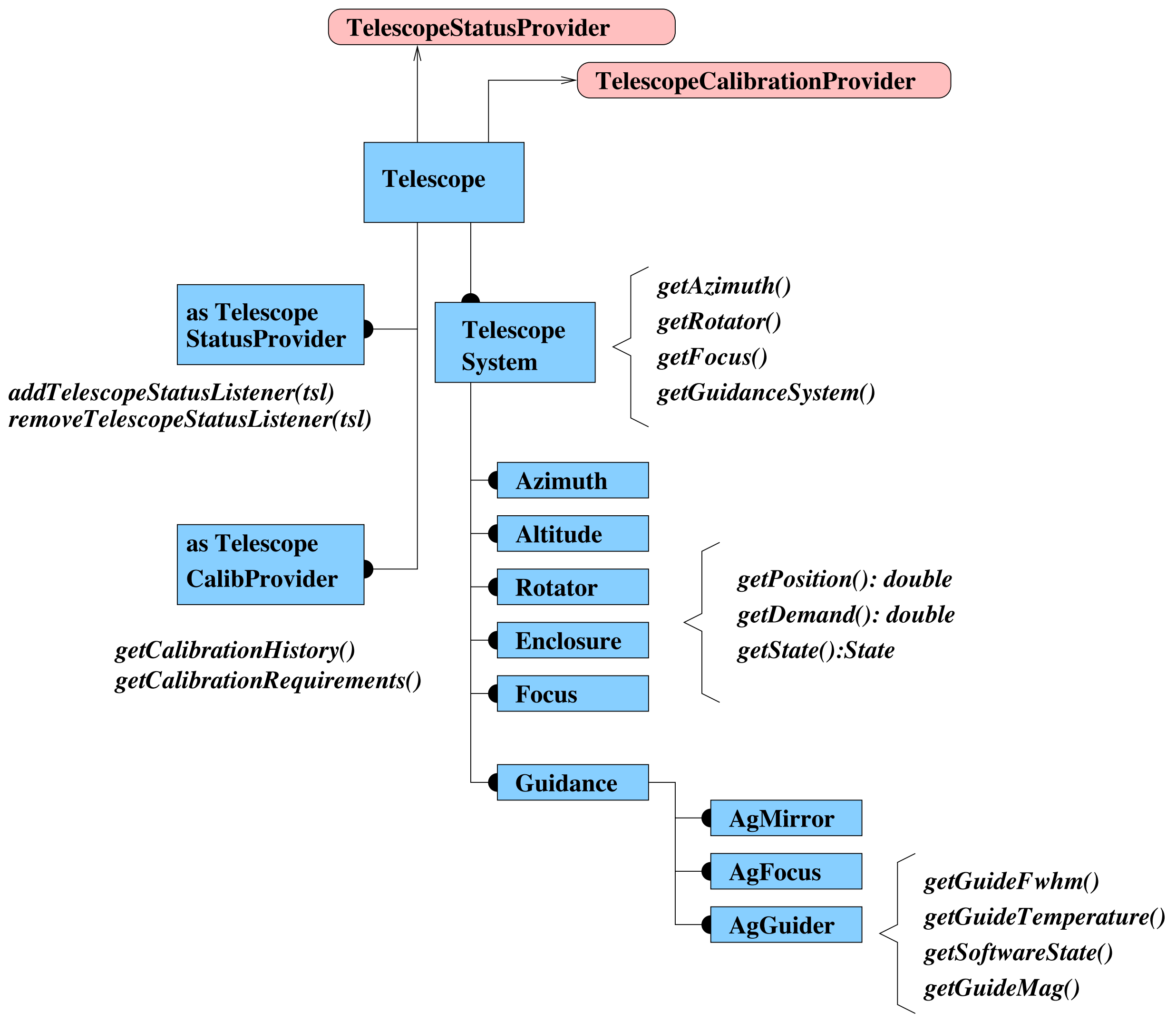
CIL Proxy layer



TCM

- Main access point interface is *Telescope* implemented by *ngat.tcm.BasicTelescope*.
- Provides model of system *TelescopeSystem*.
  - Axes, enclosure, focus, guidance, payload.
- Access hook for attaching status listeners
  - *TelescopeStatusProvider* and *TelescopeStatusListener*
- Provides calibration requirements and stores calibration history.
  - *TelescopeCalibrationProvider*
- Configured via *telescope.xml*.

TELESCOPE MODEL



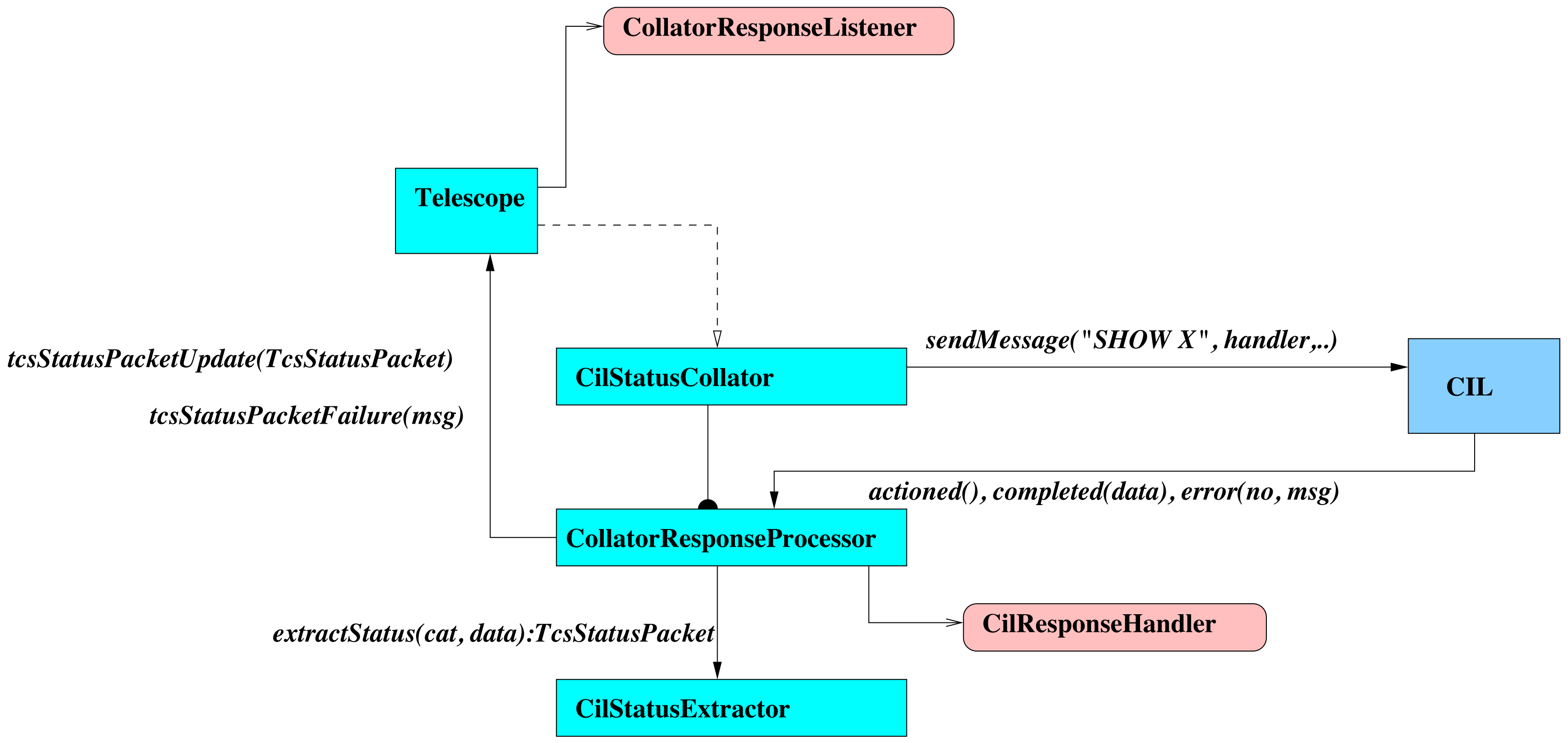
STATUS FROM TCS: OLD SYSTEM

- Old System based on classes in *ngat.rcs.scm*.
- A class *StatusPool* acts as the repository for all TCS status information.
- It holds data about each of the *segments* of the TCS status, divided up according to the logic of the available TCS SHOW commands.
- Status information is pulled from the TCS via a series of *TcsStatusClients*. These send the relevant SHOW X commands and push the returned data into *StatusPool* as *TCS_Status.Segments*.
  - Mech, State, Meteo, Astrom, Limits, Time, Autoguider…
- StatusPool then disseminates this information to other classes which have registered with it via the *java.util.Observer* interface.
  - *FitsHeaderInfo*
  - various *Monitors* used in the predecessor to ERS.

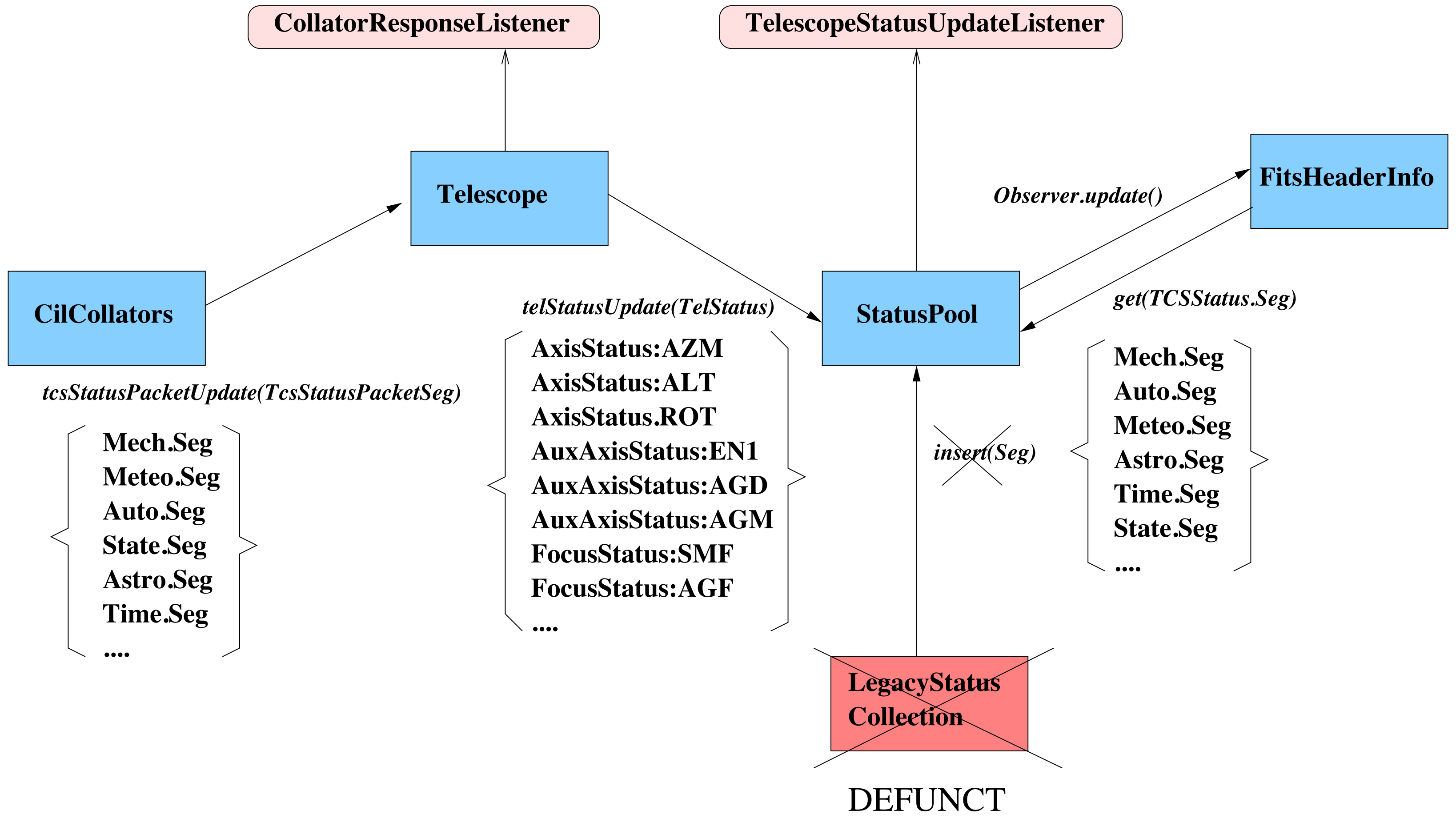
Status from TCS: New system

- Based on *ngat.tcm*.
- Various Collators (one per TCS status category) send SHOW X commands to TCS via CIL service.
- A *CollatorResponseProcessor* acts as handler for the returned CIL completed message.
- It translates the content into a *TCSStatusPacket Segment* representing the appropriate TCS status category.
  - Mech, meteo, autoguider, state, limits…
- These are passed to the Telescope (as *CollatorResponseListener* via calls to *tcsStatusPacketUpdate*(*TcsStatusPacket*).
- Internally the Telescope updates its model of the *TelescopeSystem* and passes the information out to registered listeners via *TelescopeStatusListener* interface.
- The status objects sent out have been reconstructed into types which represent the logical structure of the telescope systems…
  - Azm, Alt, En1, En2, Pmc, Pms, Smf, Agf, Tim, Env, Agg, Scf…
- Telescope status is sent out to various interested parties..
  - ERS, *TrackingMonitor*, *AgMonitor*, *Tweaker*, *Livestatus*, *OpsUI* and for now *StatusPool* to allow FITS headers to update.

Collecting TCS status


CollatorResponseListener
Telescope
tcsStatusPacketUpdate(TcsStatusPacket)
tcsStatusPacketFailure(msg)
CilStatusCollator
sendMessage("SHOW X", handler,..)
CIL
actioned(), completed(data), error(no, msg)
CollatorResponseProcessor
extractStatus(cat, data):TcsStatusPacket
CilResponseHandler
CilStatusExtractor

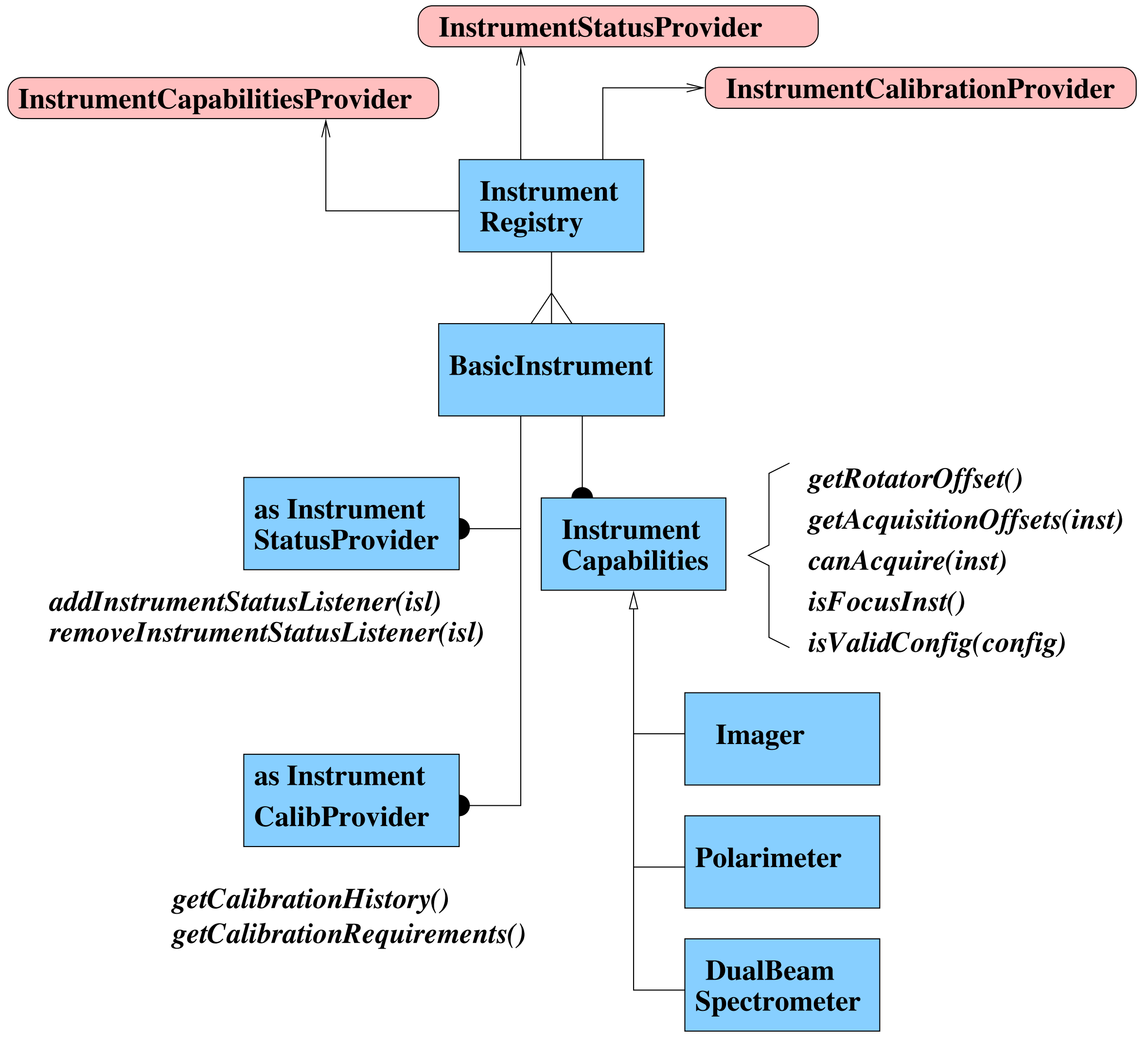
Propagation of TCS status



ICM

- Main access point interface is *InstrumentRegistry* implemented by *ngat.icm.BasicInstrumentRegistry*.
- Provides access to a list of available instruments and instrument services, which then provide..
- Access hooks for attaching status listeners
  - *InstrumentStatusProvider* and *InstrumentStatusListener*
- Calibration requirements and storing calibration history *InstrumentCalibrationProvider*.
- Capabilities providers based on *BasicInstrument* which allow properties of the instruments to be discovered..
  - getRotatorOffset() : double
  - getWavelength(InstConfig) : Wavelength
  - isValidConfig(InstConfig) : boolean
  - canAcuire(instID) : boolean
  - getAcquisitionOffsets(instID) : XPositionOffset
  - isFocusInstrument() : boolean
- Configured via *ireg.xml*.

Instrument Model


InstrumentStatusProvider
InstrumentCapabilitiesProvider
InstrumentCalibrationProvider
Instrument Registry
BasicInstrument
as Instrument StatusProvider
Instrument Capabilities
getRotatorOffset()
getAcquisitionOffsets(inst)
canAcquire(inst)
isFocusInst()
isValidConfig(config)
addInstrumentStatusListener(isl)
removeInstrumentStatusListener(isl)
Imager
as Instrument CalibProvider
Polarimeter
DualBeam Spectrometer
getCalibrationHistory()
getCalibrationRequirements()

Instrument Status

- Instrument status is collected by the object modeling the instrument within the Instrument registry (ireg).
  - Imager, Polarimeter, DualbeamSpec, TipTiltImager..
- Uses an internal monitor thread per instrument which send out GET_STATUS commands and processes the results into an *InstrumentStatus*.
- Interested parties can subscribe to the status feed by *InstrumentStatusProvider* interface…
  - Scheduler, OpsUI, LiveStatusProviders…
- Many tasks within TMS need instrument status but they are ephemeral so not worth subscribing to feeds. Access models directly…

```
// get offsets for acquiring FRODO from IO:O
InstDesc ioID  = ireg.getDescriptor("IO:O");
InstrumentCapabilitiesProvider iop = ireg.getCapabailitiesProvider(ioID);
InstrumentCapabilities io = iop.getCapabilities();  // instance of ngat.icm.Imager
InstDesc frodoID  = ireg.getDescriptor("FRODO");
IPositionOffset io_acq_frodo = io.getAcquisitionOffsets(frodoID);

// find out if IO:O is online and not reporting as faulty
InstrumentStatusProvider iosp = ireg.getStatusProvider(ioID);
InstrumentStatus iostat = iosp.getStatus();
boolean io_is_ok = iostat.isOnline() && iostat.isFunctional();
```

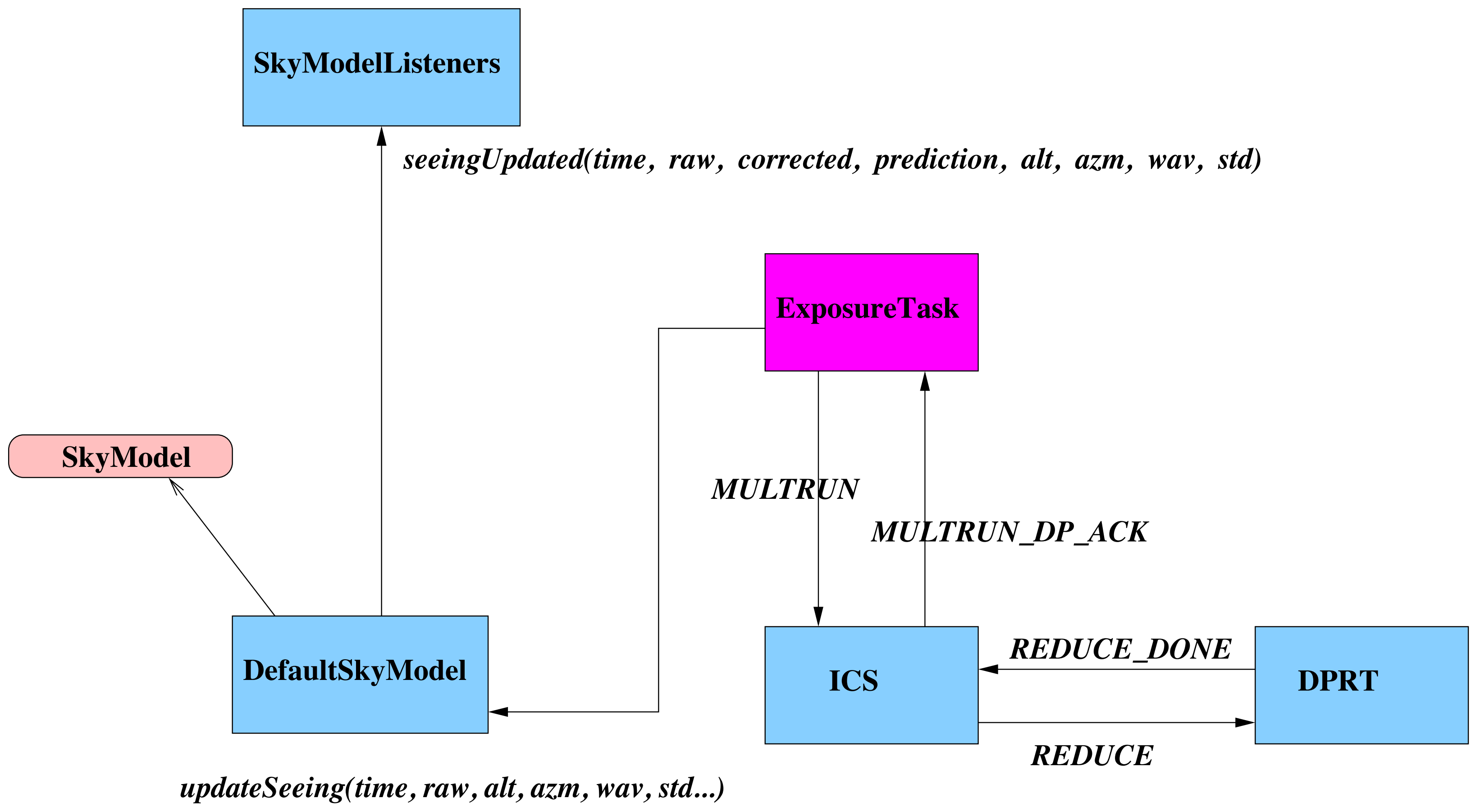
EMS

- Main access points are:
  - *SkyModel* – seeing and photometricity.
  - *MeteoProvider* – weather from various sources.
- *SkyModel* is updated using seeing extracted from reduced images. RATCam, IO:O, future…
- *MeteoProvider* is updated using information provided by the TCS, BCS cloud camera and via the TNG Dust monitor website.

SkyModel

- When an exposure is made using an instrument which is linked to the DPRT the *ExposureTask* receives updates:-
  - When an image (i) within a multrun is taken the ExposureTask receives notification via a *MULTRUN_ACK* which allows the telescope's altitude alt_i to be recorded along with the time of the actual exposure t_i.
  - At the end of the multrun each image is reduced and the ExposureTask receives *MULTRUN_DP_ACKs* containing the seeing s_i.
  - The wavelength wav_i of the exposure is obtained from the *InstCapabilities* of the instrument using e.g. *Imager.getWavelength*(*currentConfig*)
- SkyModel is then updated using a call to:-
  - SkyModel.updateSeeing(t_i, s_i, alt_i, azm_i, wav_i…)
- The SkyModel processes the update, rejecting seeing which is too large, ignoring updates which are from non-standards unless they improve the prediction, correcting for altitude and wavelength and finally performing an exponential weighted average.
- There are exceedingly detailed notes on the calculation and much more in the telescope wikiwords *SelectingSeeingCuts and LTSeeing*).

SkyModel

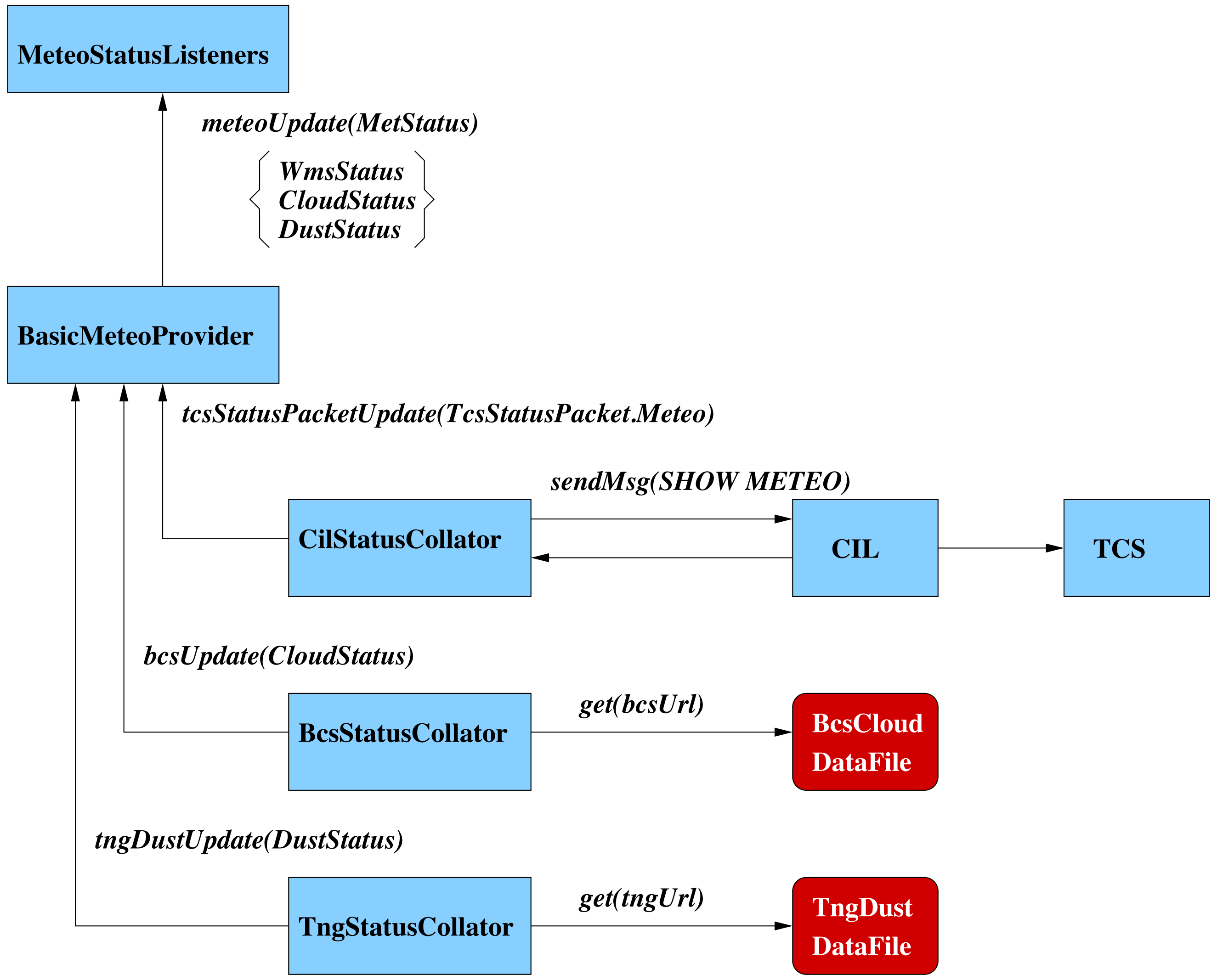


SkyModelListeners
seeingUpdated(time, raw, corrected, prediction, alt, azm, wav, std)
ExposureTask
SkyModel
MULTRUN
MULTRUN_DP_ACK
DefaultSkyModel
ICS
REDUCE_DONE
DPRT
REDUCE
updateSeeing(time, raw, alt, azm, wav, std...)

MeteoProvider

- MeteoProvider obtains details of the WMS status from the TCS via a *CilStatusCollator* similarly to the way the TCM gets TCS status updates.
- The TCS segments are re-arranged into MeteoStatus objects and sent out to any interested subscribers:-
  - ERS, OpsUI, LiveDataFeed.
- BCS cloud data and TNG Dust data are obtained via separate Collators which grab data from specific files on the occ and parse these to extract DustStatus and CloudStatus objects.
- The BCS and Dust files have previously been generated by cron scripts running on occ and ltproxy and can be found here:-
  - /occ/data/tng_dust.dat
  - /occ/data/cloud.dat

MeteoProvider



MeteoStatusListeners
meteoUpdate(MetStatus)
WmsStatus
CloudStatus
DustStatus
BasicMeteoProvider
tcsStatusPacketUpdate(TcsStatusPacket.Meteo)
sendMsg(SHOW METEO)
CilStatusCollator
CIL
TCS
bcsUpdate(CloudStatus)
BcsStatusCollator
get(bcsUrl)
BcsCloud DataFile
tngDustUpdate(DustStatus)
TngStatusCollator
get(tngUrl)
TngDust DataFile