DARC:
The Durham AO Real-time Controller

DARC practical session

Alastair Basden

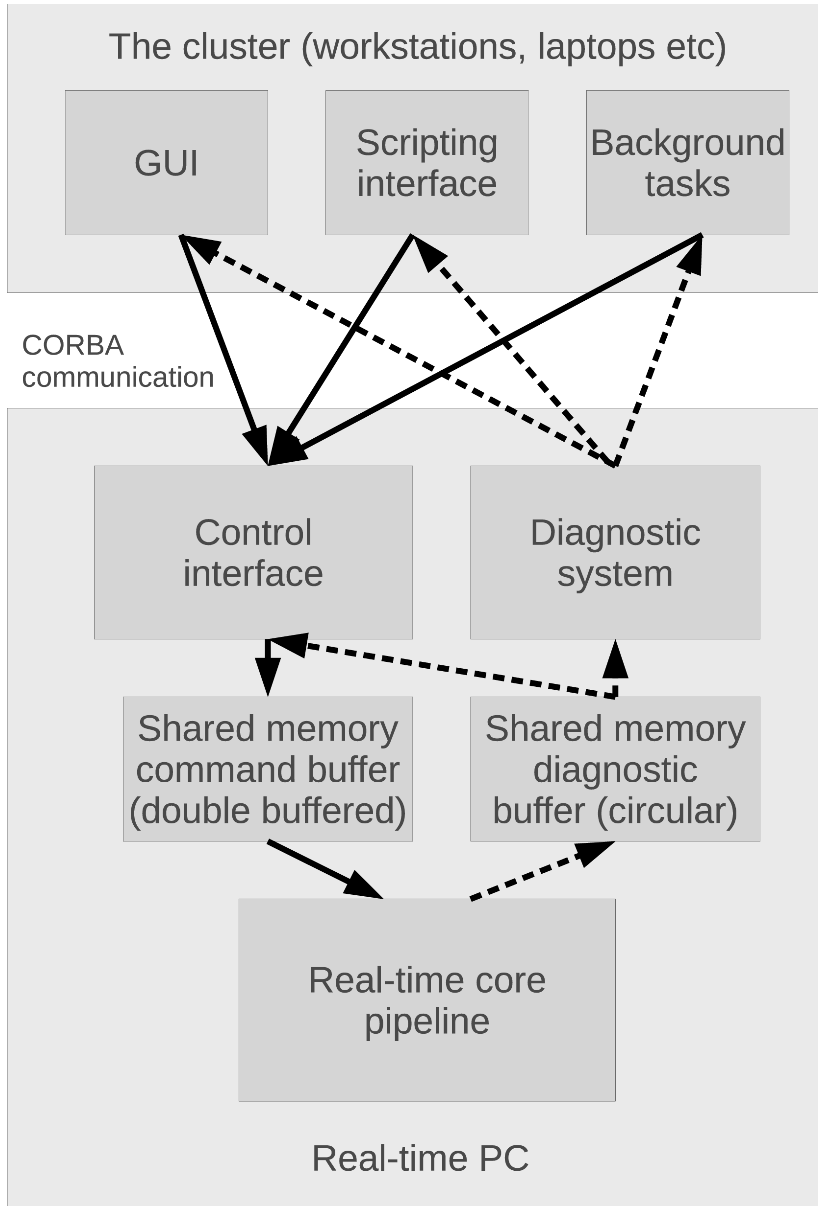
15th April 2011

Contents

DARC overview

- Open source, available from sourceforge
- GPL licensed
- PC based (unix)
  - Suitable for any 8-10m class telescope AO system
- Easy to install and develop
- Highly configurable
  - Easy to adapt for different camera and mirror configurations
- Modular design
  - Hardware modules possible
    - e.g. GPU reconstruction on-sky
  - Interfaces well defined
- Shared memory double buffered parameter buffer
  - Control can be decoupled from the real-time pipeline
- Shared memory telemetry interface
  - Telemetry decoupled from the real-time pipeline
  - Telemetry bottlenecks do not slow down or halt the RTCS
- Controlled via middleware (CORBA)
  - Though this can easily be changed to suit other requirements

DARC components

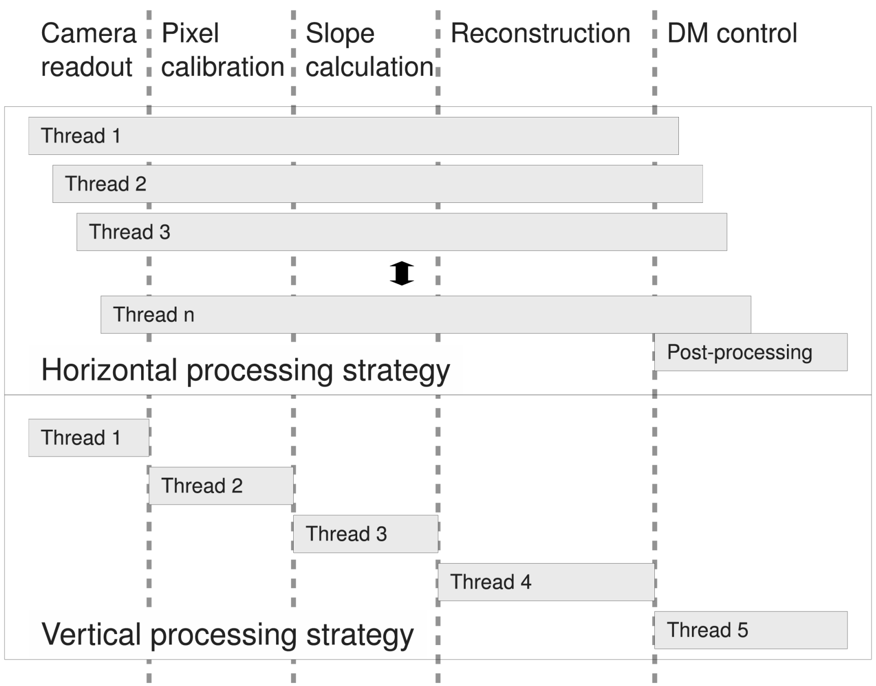


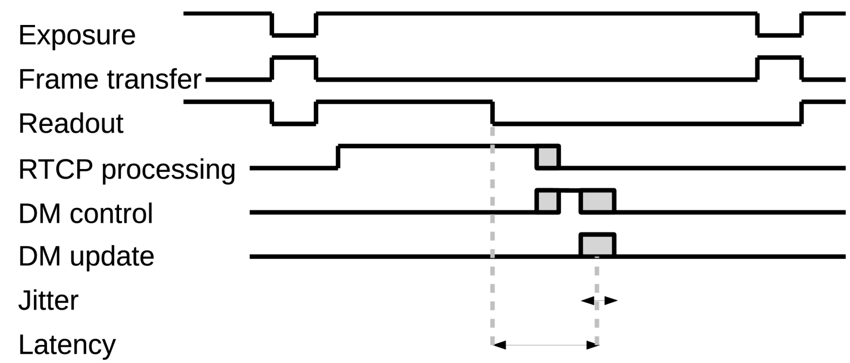
DARC architecture

- Multi-threaded (and optionally multi-process)
- Configurable
  - User defined number of threads
  - User defined thread priorities and affinities
  - Allows optimum performance to be achieved
    - Many user options to tailor performance for a given AO system
- Horizontal processing strategy
  - Each thread performs multiple operations (start to finish)
  - Lower latency, fewer synchronisation primitives required

Processing strategy

- Processing begins as soon as first pixels become available
  - Calibration, slope calculation and partial reconstruction





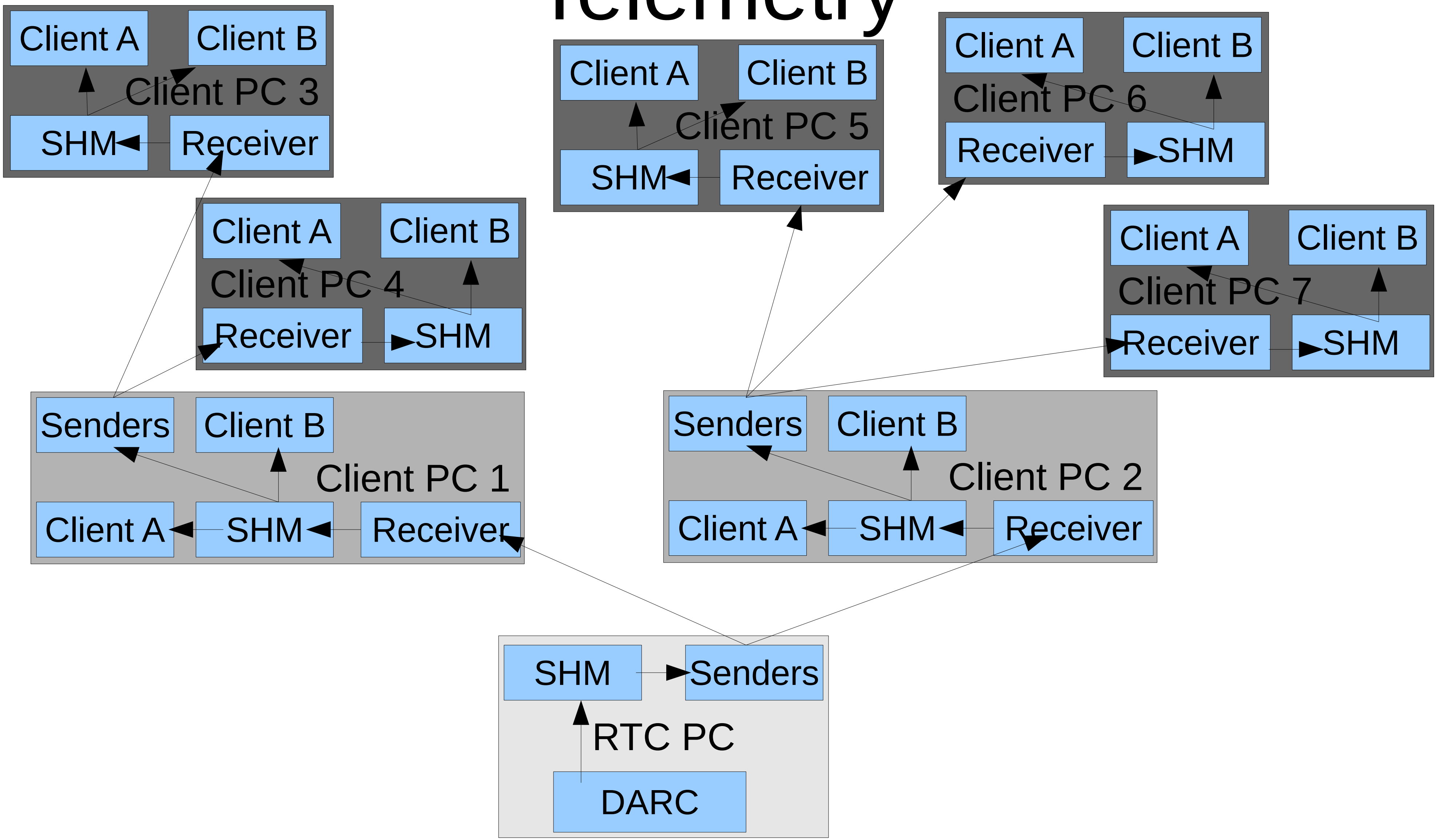
DARC Telemetry

- Telemetry outputs include (as separate streams)
  - Raw and calibrated pixels (rtcPxlBuf, rtcCalPxlBuf)
  - Slopes (rtcCentBuf)
  - Phase (rtcMirrorBuf)
  - Mirror demands (rtcActuatorBuf)
  - Status (rtcStatusBuf)
  - And others
    - Adaptive window positions
    - Correlation centroiding images
    - User defined

Telemetry implementation

- DARC writes telemetry data to individual SHM circular buffers
  - One per telemetry stream
  - Optional decimation parameter (individually settable)
    - DARC will only write what you want it to
- Telemetry sent from RTCS to client nodes via TCP socket
  - With a separate data decimation for each client
    - Different clients can receive at different rates
      - e.g. logging and GU
  - On the client node, the receiver writes the data into a shared memory circular buffer
  - Data is accessed by clients on the client nodes, by reading the local shared memory circular buffer
  - Data can be sent from client nodes to other client nodes
    - Reducing DARC workload/network bandwidth
- Custom telemetry interface can also be implemented

Telemetry



Client A
Client B
Client PC 3
SHM
Receiver
Client A
Client B
Client PC 5
SHM
Receiver
Client A
Client B
Client PC 6
Receiver
SHM
Client A
Client B
Client PC 4
Receiver
SHM
Client A
Client B
Client PC 7
Receiver
SHM
Senders
Client B
Client PC 1
Client A
SHM
Receiver
Senders
Client B
Client PC 2
Client A
SHM
Receiver
SHM
Senders
RTC PC
DARC

Telemetry choice?

- Why not UDP?
  - Data loss – critical for e.g. tomography
- Why not broadcast?
  - Relies on UDP
- Why not CORBA?
  - Poor performance for data streaming
- Why not a Data Distribution Service (e.g. DDS)?
  - Too complicated – increases installation difficulty
  - Would introduce problems if one link was much slower than others
    - But a user is free to implement if they wish... (as CANARY did with CORBA)
- Why not multicast?
  - Relies on UDP

DARC installation

- Time for us to try installing DARC
  - Please follow the instructions provided

Environment variables

- Point to the CORBA name server:
  - export ORBInitRef=”NameService=corbaname::localhost”
- Set you PATH, PYTHONPATH and LB_LIBRARY_PATH variables
- The file etc/rtc.bashrc should do this for you
  - source etc/rtc.bashrc

Directory structure

bin/	Binary files
conf/	Configuration files (darc and plots)
doc/	Documentation
etc/	Example environment file
idl/	CORBA interface definition files
include/	C headers
lib/	Libraries (when built)
lib/python/	Python files
src/	C source code
test/	Files for testing

Documentation

Lets make some documentation:

cd doc

make

Running DARC

- omniNames
  - (not part of darc)
- darccontrol
  - darcmain
- darcmagic
  - darctalk
- darcgui
- darcplot

darccontrol

- Options:
  - configuration file name (.py or .fits)
  - -bBUFSIZE (in bytes)
    - To increase the shared memory parameter buffer
    - Default 64MB
    - For higher order AO systems
  - -o
    - Don't redirect stdout – so you can see the output on terminal – useful when debugging
  - -eNHDR
    - to increase the number of allowable parameters (default 128)
  - --prefix=PREFIX
    - to change the DARC prefix

Multiple instances

- Each DARC instance creates a CORBA object, and files in shared memory
- To use multiple instances of DARC simultaneously, you need to specify a prefix:
  - Start darc with darccontrol –prefix=WHATEVER
  - Then use this prefix whenever you communicate (see later)
  - This prefix prefixes anything placed in /dev/shm/
    - parameter buffers
    - telemetry buffers

DARC configuration files

- Lets look at a configuration file...
  - conf/configcamfile.py
  - Aim to populate a “control” structure with parameters
- Missing parameters?
  - Default values inserted (if known about)

DARC parameters

- What are they, what do they do?
- Lets look at doc/man.pdf

Important parameters

- ncam
- npxlx,npxly,nsub
- subapLocation
- pxlCnt
- etc

Performance optimisation

- There are lots of ways in which performance can be optimised
  - ncamThreads – specify the number of threads
  - threadPriority – the priority of these threads
  - threadAffinity – the processor(s) on which each thread is allowed to run
  - The pixel count array
    - Don't read unnecessary pixels
    - If several sub-apertures have the same pixel count value, they can be processed together
  - subapAllocation – allocate sub-apertures to threads
  - Individual module (dynamic library) flags
    - e.g. For sFPDP library – the block size and priority/affinity
  - Getting rid of unnecessary parameters
    - e.g. set flat field to None if all ones.
  - Sub-aperture sizes (e.g. reduce once the loop is closed)
  - noPrePostThread
    - if set, places post processing into a sub-aperture processing thread
    - reduces latency, but also reduces maximum achievable framerate

DARC development

- As a user you may wish to make changes
  - New interface modules
    - Cameras and DMs
  - New algorithms
  - New methods of control
    - e.g. to make compliant with observatory X
  - New methods for telemetry
    - e.g. to make compliant with observatory X

DARC modules

- Modular design
  - Allows rapid prototyping
  - Modules loaded and unloaded dynamically
    - Allows changes/new features while the AO pipeline is running
  - Front end (cameras)
    - Interface control documents
  - Calibration
  - Slope calculation
  - Reconstruction
  - Back end (mirror control)
    - Interface control documents
  - Figure sensing
  - Buffer interface
    - Change any parameter on a per-frame basis
- Fully asynchronous operation possible (cameras with different unrestricted frame rates)
  - Also allows distributed processing to be implemented
- Full GPU pipeline in development

Loading/unloading modules

- Status tells us whether a module is loaded
- Module name comes from e.g. cameraName, slopeName, reconName, calibrateName etc
- But is it loaded?
  - darcmagic get camerasOpen (or slopeOpen or...)
- So – write a new module, compile it, and load it into the RTC
  - without stopping the RTC!

Camera interfaces

- Developing camera interfaces
  - You have to produce a shared library which contains a set of defined functions
    - DARC can then read this library, and use these functions
  - defined in include/rtccamera.h

```
int camOpen(char *name,int n,int *args,paramBuf *pbuf,circBuf *rtcErrorBuf,
           char *prefix,arrayStruct *arr,void **handle,int nthreads,
           unsigned int frameno,unsigned int **camframeno,int *camframenoSize,
           int npxls,int ncam,int *pxlx,int* pxly);

int camClose(void **camHandle);

//subap thread (once)
int camNewFrameSync(void *camHandle,unsigned int thisiter,double timestamp);

//non-subap thread (once)
int camNewFrame(void *camHandle,unsigned int thisiter,double timestamp);

//subap thread (once per thread)
int camStartFrame(void *camHandle,int cam,int threadno);

//subap thread (lots of times)
int camWaitPixels(int n,int cam,void *camHandle);

//called when ever a parameter swap is requested (subap thread, once)
int camNewParam(void *camHandle,paramBuf *pbuf,unsigned int frameno,
              arrayStruct *arr);

//subap thread (once per thread)
int camEndFrame(void *camHandle,int cam,int threadno,int err);

//subap thread (once)
int camFrameFinishedSync(void *camHandle,int err,int forcewrite);

//non-subap thread (once)
int camFrameFinished(void *camHandle,int err);

//called if the loop is open (non-subap thread)
int camOpenLoop(void *camHandle);

//non-subap thread (once)
int camComplete(void *camHandle);
```

Time to panic?

- The good news: you don't have to create all of these
  - Only 2 are required: camOpen and camClose
  - Others are there to help if you wish
    - A minimum of 1 extra to do something useful
  - libcamfile.so uses only camNewFrameSync
    - And this would be sufficient (though maybe not quite optimal) for all full frame imagers
      - i.e. ones where you can't access a pixel stream, only a full frame
    - see src/camfile.c

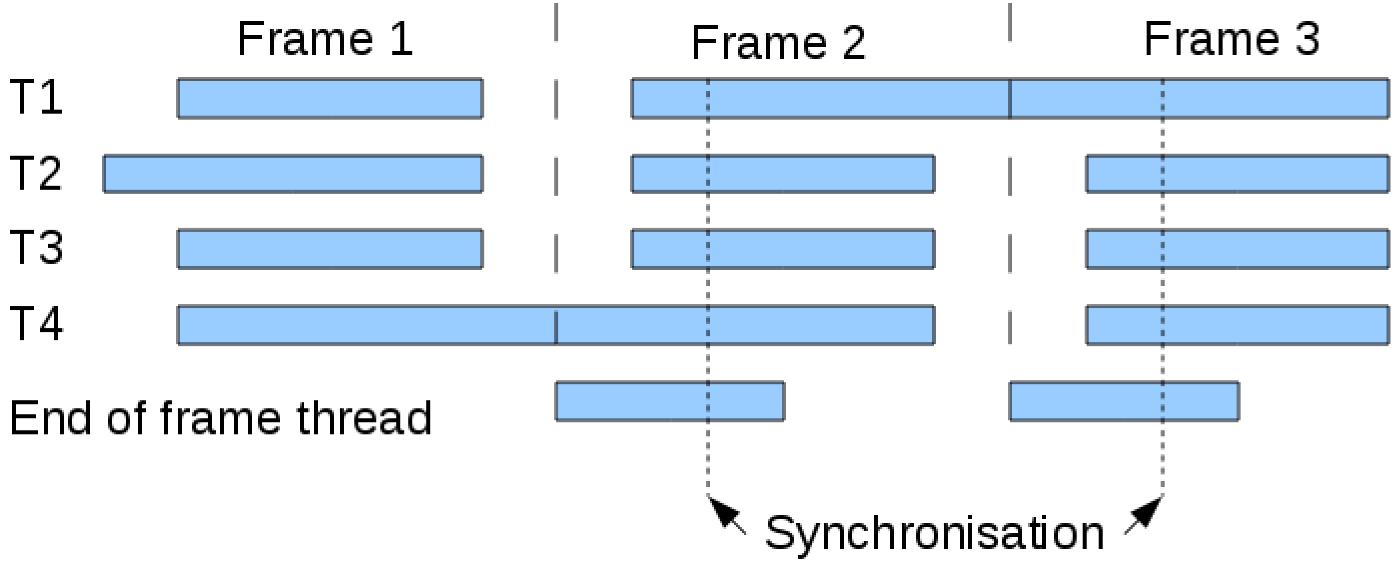
A more advanced camera

- jaicam.cpp
  - For driving a JAI Pulnix camera
    - gigE vision, >1kHz frame rates
    - CANARY figure sensor
  - Delays waiting for pixels for as long as possible
    - Waiting for pixels is what the RTC will spend a lot of its time doing
    - So use this time for other things...
- sl240Int32cam.c
  - For reading a sFPDP pixel stream
  - CANARY WFSs

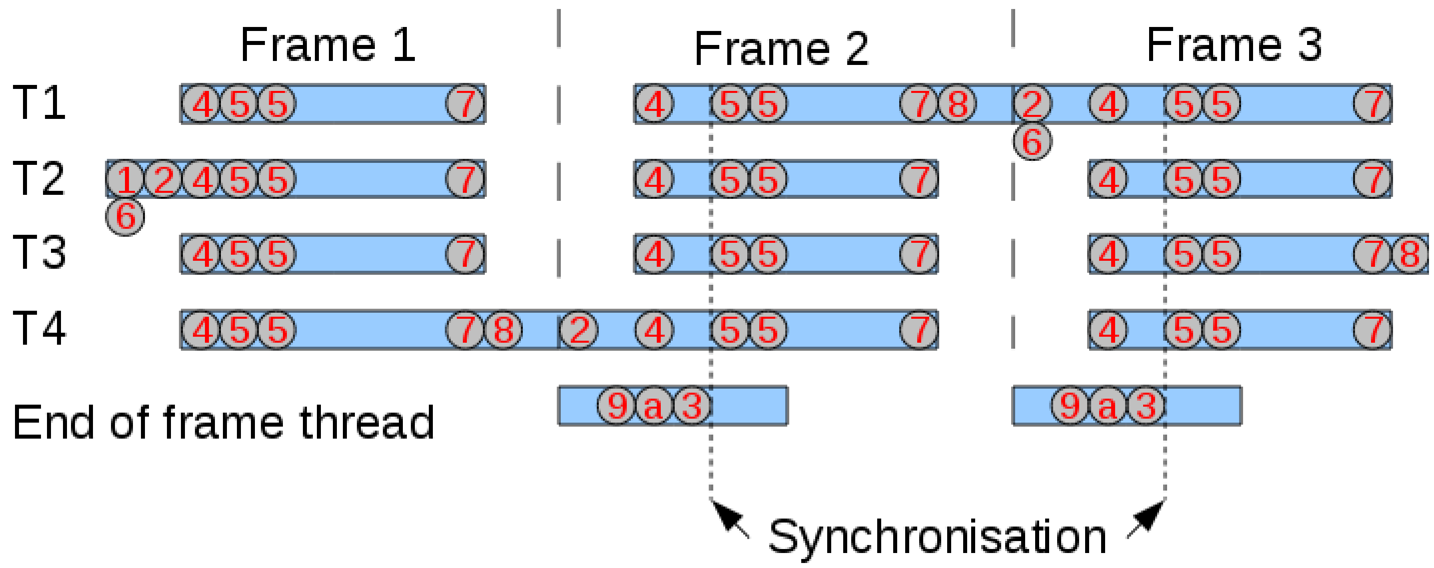
Library hook points

- Lets look at the DARC thread structure

  (NOTE: noPrePostThread option)



And when the functions are called



1 Open
2 NewFrameSync
3 NewFrame
4 StartFrame
5 camWaitPixels (or calibrateNewSubap or slopeCalcSlope or reconNewSlopes)
6 NewParam
7 EndFrame
8 FrameFinishedSync
9 FrameFinished
a Complete

Function args

```
int camOpen(char *name,int n,int *args,paramBuf *pbuf,circBuf *rtcErrorBuf,
        char *prefix,arrayStruct *arr,void **handle,int nthreads,
        unsigned int frameno,unsigned int **camframeno,int *camframenoSize,
        int npxls,int ncam,int *pxlx,int* pxly);

int camClose(void **camHandle);
```

- name – the name of the .so library (not usually used)
- n, args – Initialisation arguments are passed in as an int array with n elements
  - can be typecast to whatever you like by the library
    - e.g. camfile typecasts to char* to get a filename
- paramBuf – pointer to the current parameter buffer
- rtcErrorBuf – for writing errors too
- prefix – the darc prefix
- arr – an arrayStruct object – pointers to lots of useful arrays
- handle - the library will initialise this, which is then passed to every other camera library function call
- nthread – number of threads
- frameno – the current RTC frame number
- camframeno – can be allocated by the library into which camera frame numbers (or anything else) can be placed – and shows up in the status telemetry stream
- camframenosize – number of elements of camframeno
- npxls – total number of pixels
- ncam – number of cameras to open
- npxlx, npxly – image width/height for each camera (array of size ncam)

Function args

```
int camNewFrameSync(void *camHandle,unsigned int thisiter,double timestamp);
```

- camHandle – allocated by camOpen
- thisiter – current DARC iteration
- timestamp – current DARC timestamp

```
int camWaitPixels(int n,int cam,void *camHandle);
```

- n – number of pixels that must have arrived for this frame before returning
- cam – camera number (0 for single camera configurations
- camHandle – allocated by camOpen

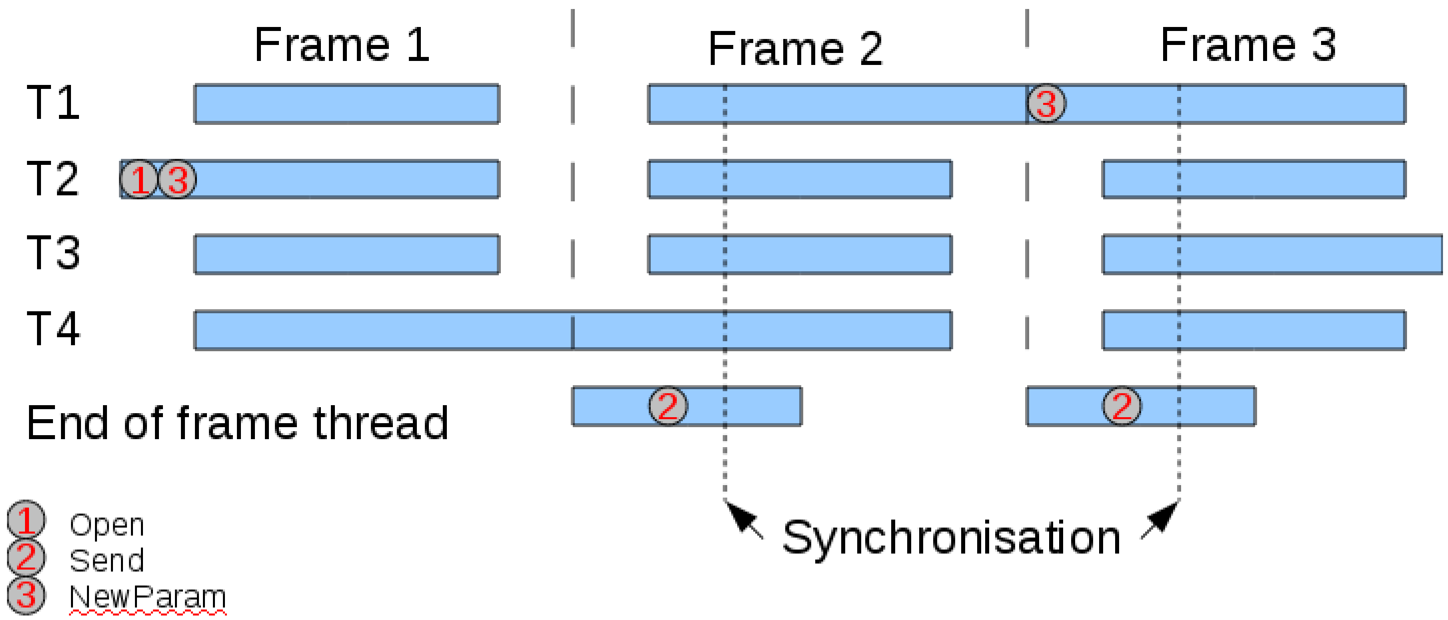
DM interfaces

- defined in include/rtcmirror.h

```
int mirrorOpen(char *name,int n,int *args,paramBuf *pbuf,circBuf *rtcErrorBuf,
           char *prefix,arrayStruct *arr,void **handle,int nacts,circBuf *rtcActuatorBuf,
           unsigned int frameno,unsigned int **mirrorframeno,int *mirrorframenoSize);
int mirrorClose(void **mirrorHandle);
int mirrorSend(void *mirrorHandle,int n,float *data,unsigned int frameno,
           double timestamp,int err);
int mirrorNewParam(void *mirrorHandle,paramBuf *pbuf,unsigned int frameno,
              arrayStruct *arr);
```

- mirrorOpen – parameters as for camOpen
- mirrorSend – n is the number of actuators which are stored in data.
  - err will be set if the data is invalid (for example partial frame/missing pixels received from camera)
    - The mirror library can then choose not to send the actuators, but can update any internal state (for more advanced algorithms)

DM Library hook points



The parameter buffer

- Used to control DARC – stores all the paramters
  - Changed by the control object (double buffered)
- Not usually required for camera libraries
  - though maybe, e.g. to change exposure time etc
- But is typically required for mirror interfaces
  - For clipping arrays, scaling and offset arrays.
- So, lets have a look
  - include/buffer.h

Parameter buffer functions

```
char *bufferMakeNames(int n,...);

int bufferGetIndex(paramBuf *pbuf,int n,char *paramList,int *index,void **values,
                   char *dtype, int *nbytes);
```

- bufferMakeNames – called with a list of n alphabetically sorted strings, that are the names of parameters required by the library
  - e.g. bufferMakeNames(5,”actMax”,”actMin”,”actOffset”,”actScale”,”nacts”)
  - This is called once only, during mirrorOpen
  - The return value (paramList) is then used in bufferGetIndex
- bufferGetIndex is called each time a parameter swap is requested
  - Runs through the list of available parameters (in shared memory)
    - stores those requested in bufferMakeNames
      - The index in the parameter buffer
      - The pointer to actual parameter value (typecast to void*)
      - The datatype of this parameter (single character)
      - The number of bytes for this parameter
    - The library should then check that the parameter is of correct type and size, and then use it...

Buffer example (from mirrorSocket.c)

```
//Somewhere near the top of the file, define what parameters are needed...
typedef enum{
  MIRRORACTMAX,
  MIRRORACTMIN,
  MIRRORACTOFFSET,
  MIRRORACTSCALE,
  MIRRORNACTS,
  //Add more before this line.
  MIRRORNBUFFERVARIABLES//equal to number of entries in the enum
}MIRRORBUFFERVARIABLEINDX;
//Note, the enum and parameter names are alphabetical.
#define makeParamNames() bufferMakeNames(MIRRORNBUFFERVARIABLES,    \
                                         "actMax",                  \
                                         "actMin",                  \
                                         "actOffset",               \
                                         "actScale",                \
                                         "nacts"    )

//Then, in function mirrorOpen()
{
  mirrorStruct->paramNames=makeParamNames();
}

//Then in function mirrorNewParam(...,paramBuf *pbuf,...)
{
  int indx[MIRRORNBUFFERVARIABLES];
  int nbytes[MIRRORNBUFFERVARIABLES];
  void* values[MIRRORNBUFFERVARIABLES];
  char dtype[MIRRORNBUFFERVARIABLES];
  int nfound=bufferGetIndex(pbuf,MIRRORNBUFFERVARIABLES,
                            mirrorStruct->paramNames,indx,values,dtype,nbytes);
  //Then use the parameters...
  if(nbytes[MIRRORACTMIN]==sizeof(unsigned short)*mirrorStruct->nacts &&
     dtype[MIRRORACTMIN]=='H')
    mirrorStruct->actMin=(unsigned short*)values[MIRRORACTMIN];
}
```

DM examples

- src/mirrorSocket.c
  - Sends actuator values to a TCP/IP socket
- src/mirrorSHM.c
  - Writes actuator values to shared memory
- src/mirrorSL240.c
  - Writes actuator values to sFPDP
- src/mirrorPdAO32.c
  - Writes actuator values to a DAC card
  - Note the removal if statements from loops where possible to improve performance

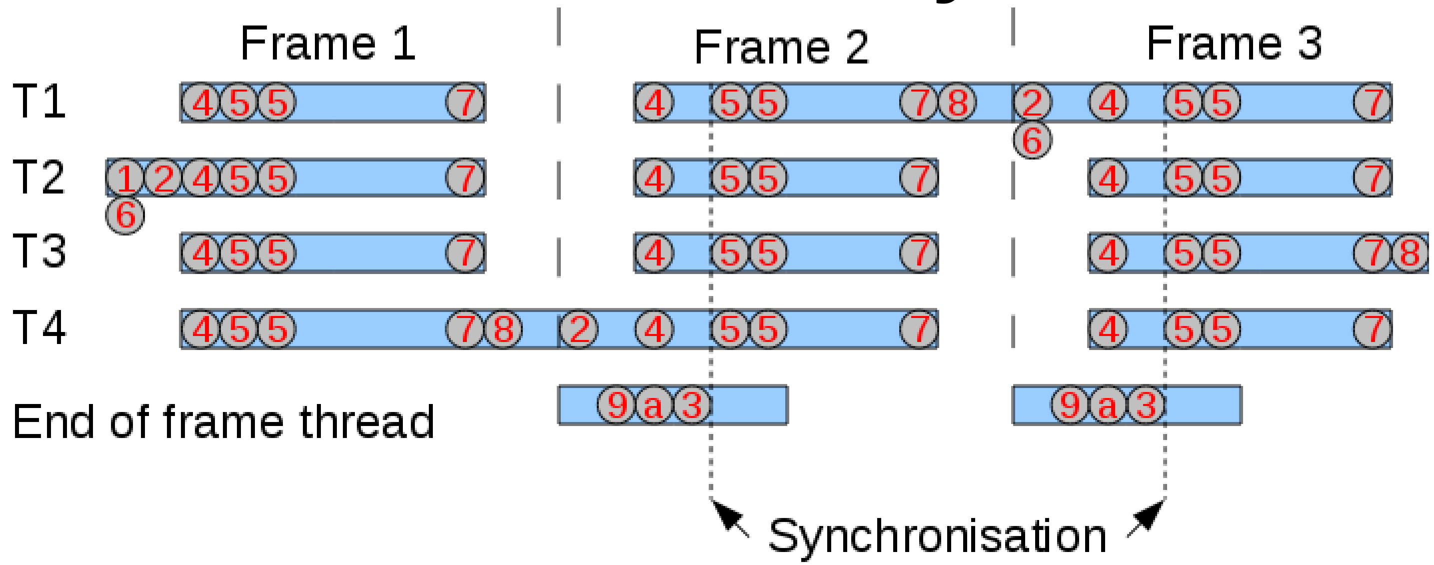
Algorithm development

- You may wish to develop your own algorithms
  - These may be adaptations of something existing
    - e.g. a new image thresholding routine
    - Change an existing library
  - Or completely different
    - e.g. a new way of doing wavefront reconstruction
    - Write a new library
  - These can then be inserted into the DARC main development tree for others to download
    - via the maintainer (i.e. me)

Image calibration

- Currently, only one library exists for image calibration
  - src/rtccalibrate.c
- New libraries can use any or all of the function declarations found in include/rtccalibrate.h
- These are very similar to those for the camera library

Calibration library hooks



1 Open
2 NewFrameSync
3 NewFrame
4 StartFrame
5 camWaitPixels (or calibrateNewSubap or slopeCalcSlope or reconNewSlopes)
6 NewParam
7 EndFrame
8 FrameFinishedSync
9 FrameFinished
a Complete

calibrateNewSubap

```
int calibrateNewSubap(void *calibrateHandle,int cam,int threadno,int cursubindx,
        float **subap,int *subapSize,int *curnpxlx,int *curnpxly)
```

- threadno – the thread number that is calling this function
- cursubindx – current sub-aperture index
- subap – pointer to the temporary subaperture workspace
  - The calibrated sub-aperture data is to be placed in here.
  - Can be increased in size if necessary by this function
- subapSize – the number of elements allocated in subap
  - Update if necessary (if this is found not to be enough)
- curnpxlx, curnpxly – to be updated with the current number of pixels (x,y) in this sub-aperture
- The raw pixel data is obtained from the arrayStruct provided to the calibrateOpen function

The default calibration library

- Uses calibrateOpen, calibrateClose, calibrateNewParam, calibrateNewSubap
- When calibrateNewSubap is called:
  - computes size of sub-aperture
    - using subapLocation parameter (actually an internal version to allow for moving windows)
  - Copies the raw camera data required for this subaperture into “subap”
    - After conversion to float32
  - Then proceeds to calibrate “subap” (dark subtraction, flat field, background subtraction, thresholding, brightest pixel selection, raising to power)
  - Then copies the calibrated data back into a global calibrated image array (to be used for telemetry, but not internal processing)

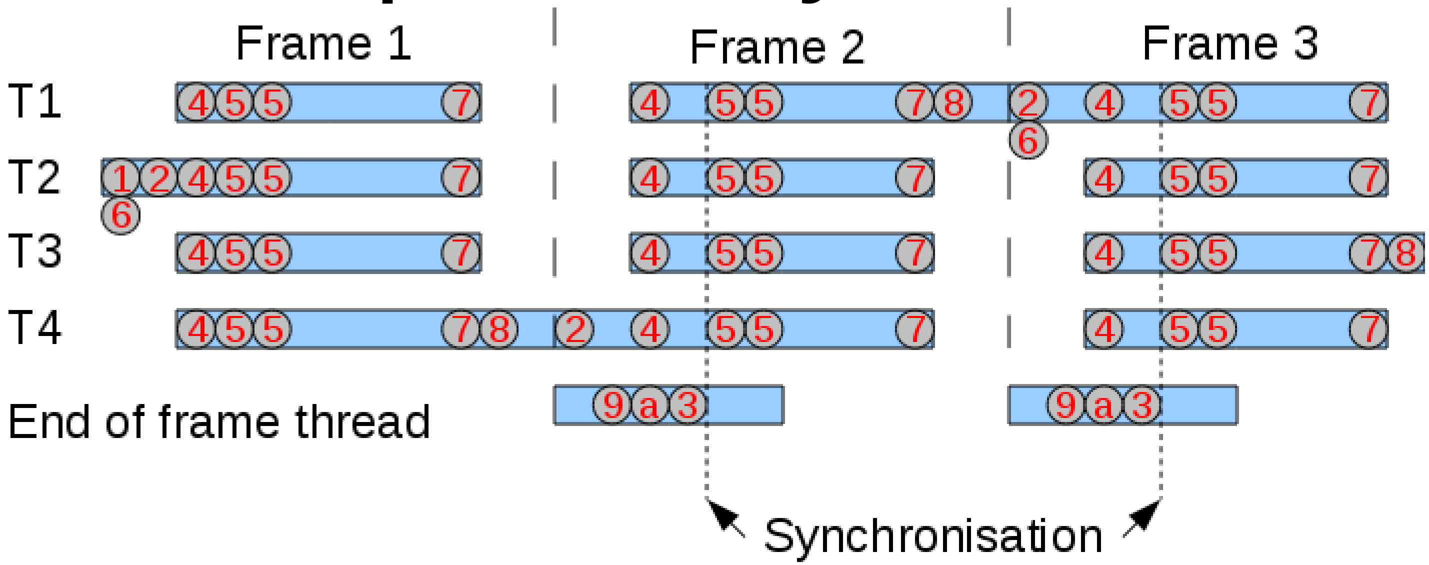
Changes?

- On the to-do list:
  - Add support for arbitrary pixel assignment to this library (already supported in DARC)

Slope computation

- Currently, DARC is focussed on Shack-Hartmann sensors
  - But the flexibility also allows it to be used with pyramid sensors without change
- Other sensor types would require new libraries developing
  - e.g. curvature, etc
- librtcslope.so, src/rtcslope.c
- Header definitions found in include/rtcslope.h
  - Very similar to those in the camera library

Slope library hooks



1. Open
2. NewFrameSync
3. NewFrame
4. StartFrame
5. camWaitPixels (or calibrateNewSubap or slopeCalcSlope or reconNewSlopes)
6. NewParam
7. EndFrame
8. FrameFinishedSync
9. FrameFinished
a. Complete

slopeCalcSlope

```
int slopeCalcSlope(void *slopeHandle,int cam,int threadno,int nsubs,
                   float *subap, int subapSize,int subindx,int slopeindx,
                   int curnpxlx,int curnpxly);
```

- nsubs – the total number of sub-apertures that should have arrived before continuing
  - Useful if have a external slope calculator (e.g. SPARTA)
- subap, subapSize – created by the calibration library, containing calibrated pixel data for this sub-aperture
- subindx – the sub-aperture number to process (this index includes unused sub-apertures)
- slopeindx – the resulting centroid index (this index counts used sub-apertures only)
- curnpxlx, curnpxly – dimensions of the current sub-aperture

The default slope library

- Uses slopeOpen, slopeNewParam, slopeClose, slopeNewFrameSync, slopeCalcSlope, slopeFrameFinishedSync, slopeComplete
- slopeNewFrameSync – to update sub-aperture locations if adaptive windowing
- slopeCalcSlope – calculates the slope of the given sub-aperture
  - Places results into the arrayStruct->centroids and arrayStruct->flux
- slopeFrameFinishedSync – to update global adaptive windows
- slopeComplete – to save the correlation image (if used)
  - It could be done earlier, for example in slopeFrameFinishedSync, but do it here, to reduce latency

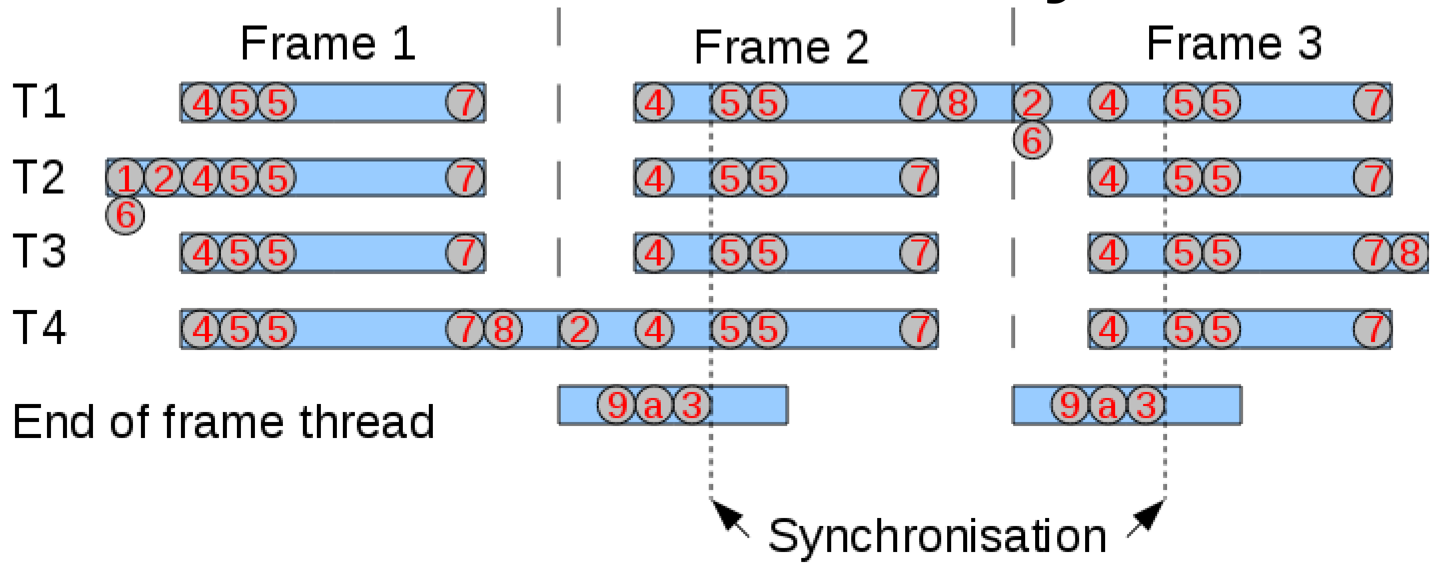
Changes?

- On the to-do list:
  - Add support for arbitrary pixel assignment

Wavefront reconstruction

- Current libraries exist for MVM reconstruction only
  - Best performance for non-ELT scale systems
- Other algorithms would require new library development
  - e.g. PCG, Fourier based reconstruction etc
- Header definitions found in include/rtcrecon.h
  - very similar to those found in the camera library

Reconstruction library hooks



1. Open
2. NewFrameSync
3. NewFrame
4. StartFrame
5. camWaitPixels (or calibrateNewSubap or slopeCalcSlope or reconNewSlopes)
6. NewParam
7. EndFrame
8. FrameFinishedSync
9. FrameFinished
a. Complete

reconNewSlopes

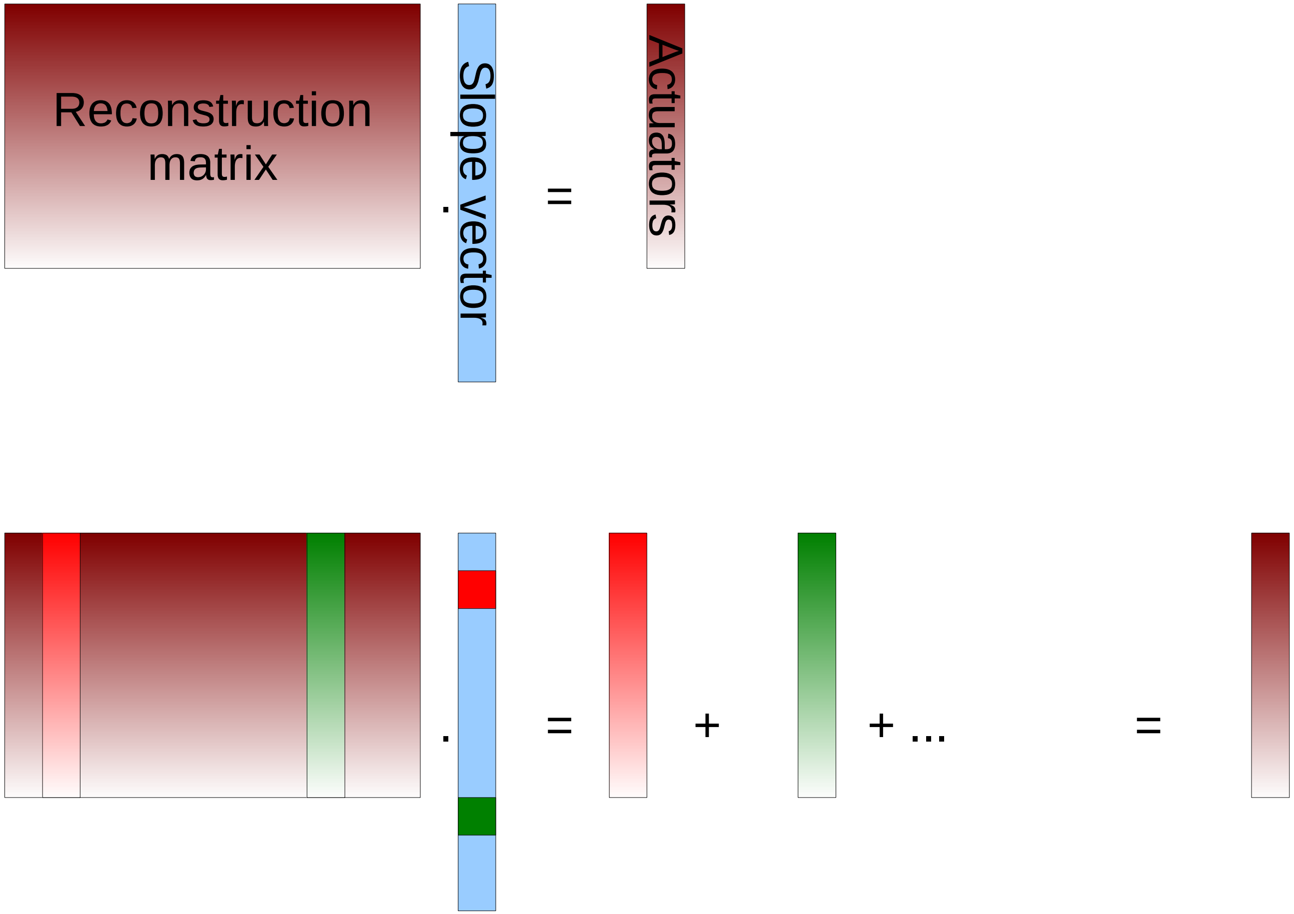
```
int reconNewSlopes(void *reconHandle,int cam,int centindx,int threadno,
                   int nsubapsDoing)
```

- cam – camera number for this thread
- centindx – slope index arriving (counting used sub-apertures only)
- threadno – thread number
- nsubapsDoing – the number of sub-apertures for which slopes are ready.

src/reconmvm.c

- Includes (by #defines) a CUDA GPU implementation
- Uses:
  - reconOpen, reconClose, reconNewParam
  - reconNewFrame
    - Initialises the state (e.g. gets the integrator ready, etc)
  - reconStartFrame
    - Per thread initialisation – clears some scratch memory
  - reconNewSlopes
    - Performs a partial MVM (a few rows of matrix with a few slope measurements)
  - reconEndFrame
    - Adds together the partial reconstructed wavefront vectors – to give the final vector once this has been called by all threads
  - reconFrameFinishedSync
    - Housekeeping
  - reconFrameFinished
    - Bleeding
  - reconOpenLoop
    - Reinitialises the state vector (i.e. to midrange) if the loop is open
- Of course, you could get away with less, at the expense of latency

Partial reconstruction



src/reconKalman.c

- Kalman filtering module
- reconOpen, reconClose, reconNewParam
- reconNewFrame
  - Updates the Kalman state vector
- reconStartFrame
  - Clears some per-thread memory
- reconNewSlopes
  - Does a row of the MVM (to get a partial result)
- reconEndFrame
  - Sums the partial results
- reconFrameFinishedSync
  - Housekeeping
- reconFrameFinished
  - Produces DM demands from phase (if necessary)
  - Bleeding
- reconOpenLoop
  - Resets the Kalman state vector

src/reconAsync.c

- We'll come to this one later...

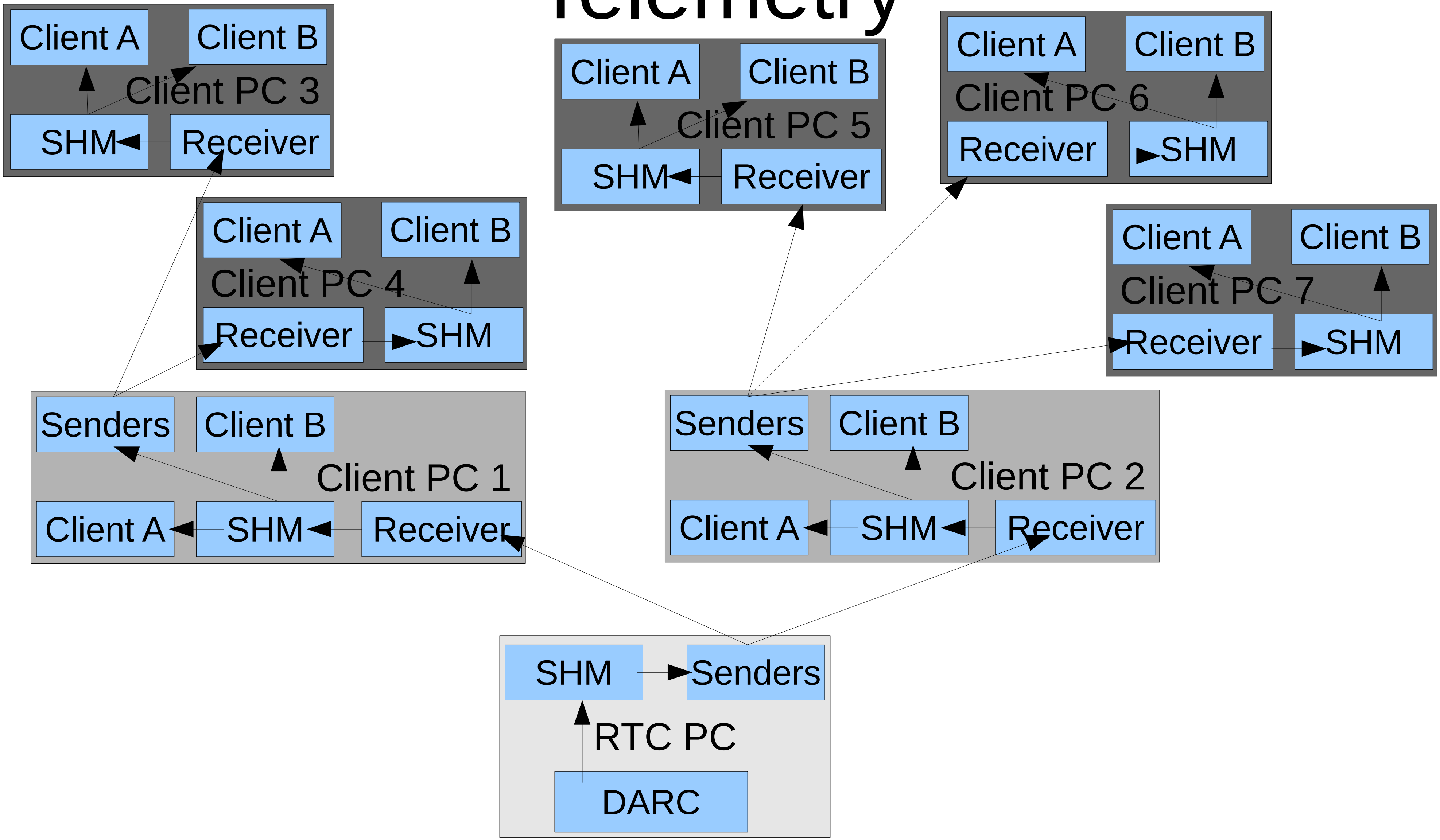
Telemetry development

- The nitty-gritty:
  - After actuators have been sent, DARC writes the current frame telemetry data to circular buffers in shared memory (if this stream is switched on)
    - At the correct decimation (i.e. not necessarily every frame)
  - For each stream, a condition variable is then signalled
  - Clients waiting for telemetry will be blocked on this condition variable
    - They then wake up, and deal with the new data
  - Circular buffers are 100 entries long by default
    - Can be changed - 100 frames gives a fraction of a second
    - Very slow clients can have their data corrupted (half an old frame, half a new frame)
    - But what is the alternative? Block the RTC? No... rather – increase the circular buffer size – or spec a more powerful computer
  - The default telemetry clients can either read from the head of the buffers
    - always taking the most recent frames – which may mean some frames are skipped
  - Or from the tail of the buffers
    - But if they fall behind by more than 75% of the buffer size, they jump back to the head

Telemetry clients

- src/sender.c
  - Reads one telemetry stream
  - Sends data to one remote client
    - src/receiver.c
    - receiver then places the data into shared memory – using a circular buffer identical to that written by DARC
    - This can then be read by local clients
- User local clients can:
  - Use the python library, lib/python/controlCorba.py, and the lower level lib/python/buffer.py
    - (see later)
  - Or use the c library, src/circ.c, include/circ.h
    - circOpenBufReader(name)
    - circGenNextFrame()
    - circGetFrame()
    - circGetLatestFrame()
- src/summer.c
  - another client – sums telemetry data, producing a new telemetry stream – which can then be sent or read as with standard DARC streams

Telemetry


Client A
Client B
Client PC 3
SHM
Receiver
Client A
Client B
Client PC 5
SHM
Receiver
Client A
Client B
Client PC 6
Receiver
SHM
Client A
Client B
Client PC 4
Receiver
SHM
Client A
Client B
Client PC 7
Receiver
SHM
Senders
Client B
Client PC 1
Client A
SHM
Receiver
Senders
Client B
Client PC 2
Client A
SHM
Receiver
SHM
Senders
RTC PC
DARC

Your own system?

- How do you provide your own telemetry system?
  - Write code using circc.c/circ.h or buffer.py/controlCorba.py
    - (or any other language for which Posix mutexes and condition variables are available)
  - Once you have a new frame of telemetry data
    - Do what you like with it
    - e.g. publish via CORBA
    - Send it over a fibre network
    - infiniband
    - etc

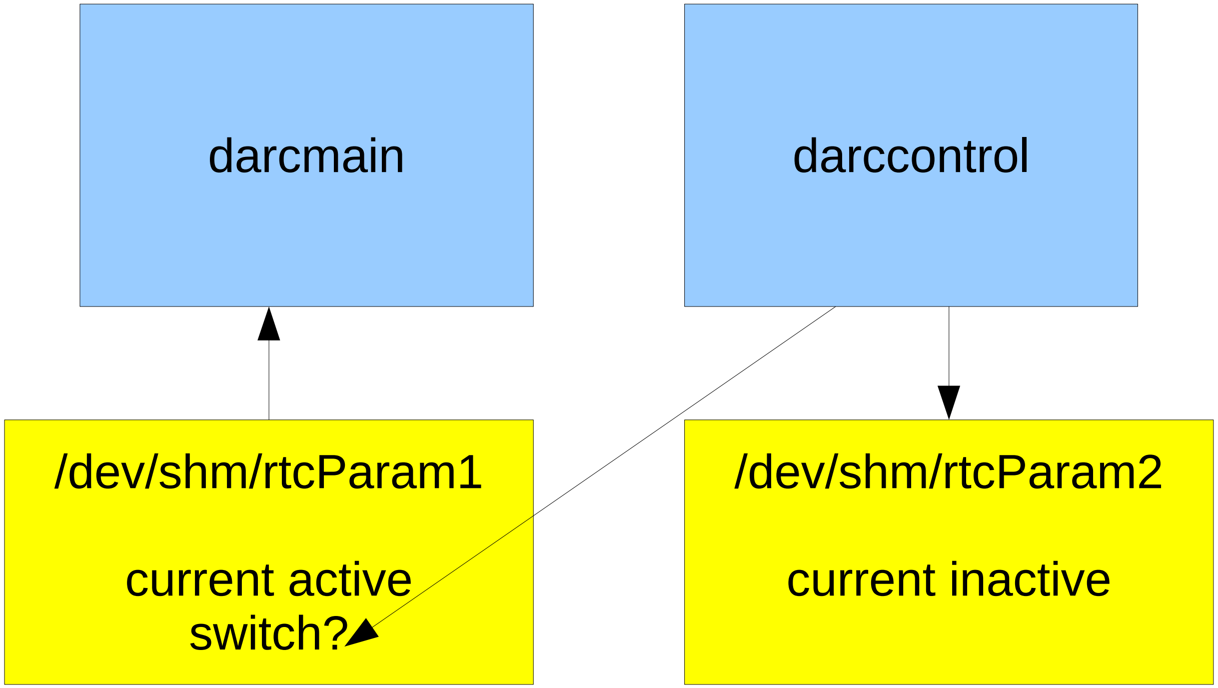
The CANARY telemetry system

- Implemented using CORBA
- Uses exactly the method mentioned previously
  - A master process sits on DARC PC
    - Clients subscribe to it
    - When new data becomes available:
      - the data is published to each client
- Written independently of DARC
  - I had nothing to do with it... - so it is usable by others!

Control overview

- The real-time core of DARC is again performed using shared memory
  - Double buffered – at any given time, one will be active (used by darcmain) and one will be for writing new data into (inactive)
    - /dev/shm/rtcParam1, /dev/shm/rtcParam2
  - When new parameters are ready in the inactive buffer
    - a flag is set in the active buffer
      - This is the only time the active buffer should be written too
    - At the start of each frame, darcmain checks this flag
      - If set, it performs a buffer swap

Buffers


darcmain
darccontrol
/dev/shm/rtcParam1
current active
switch?
/dev/shm/rtcParam2
current inactive

Parameters?

- Everything used by darcmain is a parameter:
  - A 16 byte name
  - A 1 byte data type
  - A 4 byte size
    - A potential problem for ELT systems – a trade-off to be made
      - On the todo list – implement optional large parameters
  - Other stuff too
    - A comment
    - The starting address (relative to base SHM address)
    - Dimensions and shape (not used by darcmain)
- e.g. darcmagic get -labels

Control development

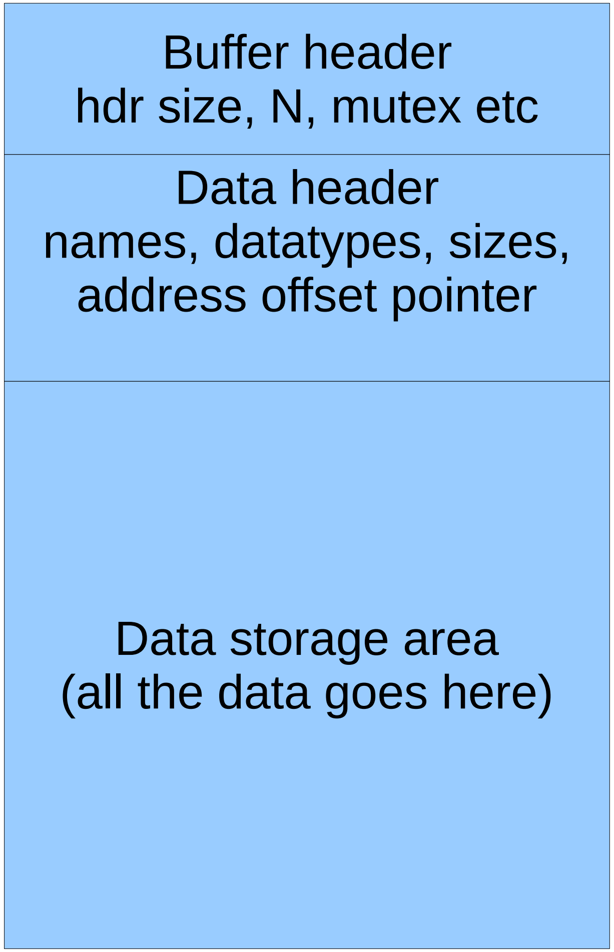
- So, if you want to develop your own control process, you can!
  - Without touching darcmain (the real-time part)
  - Your controller only has to access these buffers, and write the correct things
  - But – you would need to be careful with parameter checking
    - correct data type, correct size etc
    - otherwise darcmain will detect the error and pause itself
  - And with synchronisation (a mutex so that only one client can update buffers at a time)
- But you might be better writing a control process that talks CORBA to the default controller...
  - unless you have good reasons not too (e.g. it is too slow, or you have CORBA allergies)
    - If it is because of speed – you might consider using the buffer interface (see later) which allows changing parameters on a frame-by-frame basis guaranteed!

lib/python/buffer.py

- If you really want your own control process:
  - To write to the parameter buffers, use lib/python/buffer.py
  - Note – there is currently no .c that will write to the parameter buffers
    - It would have to implement the functionality of buffer.py

Buffer memory layout

- Each buffer is a single block of memory
- A header:
  - The header size
  - Maximum number of parameters (N)
  - flags
  - Mutex structure size
  - Condition variable size
  - A mutex
    - For obtaining the condition variable
  - A condition variable
    - Signalled by darcmain whenever a buffer swap is performed
- Then the parameter buffer
  - A data header area:
    - Parameter names[N]
    - Parameter data type[N]
    - Parameter start address offsets[N]
    - Parameter sizes[N]
    - Parameter dimensions[N]
    - Parameter comment length[N]
  - A data area:
    - The parameter data and comments



Scripting

- With Python
- With CORBA
- With command line tools
- Other languages

Python scripting

```
>>>import controlCorba
>>>ctrl=controlCorba.controlClient()
```

- You can then use ctrl to control DARC:
  - ctrl.Set(), ctrl.Get(), ctrl.GetLabels() to set and get parameters
  - ctrl.RTCinit(), ctrl.RTChalt() to initialise and stop DARC
  - ctrl.SetDecimation(), ctrl.GetDecimation() to control telemetry rates
  - ctrl.GetStream() to grab the latest data in a telemetry stream
  - ctrl.GetStreamBlock() to get multiple frames of telemetry data
  - and others
- Provides wrappers for the CORBA calls

CORBA IDL

- CORBA allows remote control of a process from almost any machine and language
- The DARC CORBA interface definition is in src/control.idl
- Can be used by any language with a CORBA implementation
- Gives a list of all the operations that can be used to control DARC
  - Most useful ones include
    - Get, Set, RTCinit, RTChalt, SetRTCDecimation, GetStreams, GetStream, GetLabels, StartStream, WatchParam, GetDecimation
  - Wrapper functions advisable
    - to convert data into datatypes expected by CORBA

Command line tools

- darcmagic and darctalk
- darcmagic is most powerful
  - gives command line control of most of the things you want
- darctalk is more restrictive
  - and has a more strict notation
    - best for inside other languages
- These are actually python processes
  - That provide a link between command line and controlCorba objects.

darcmagic

- help (or no args, or unrecognised args) – prints help message
- set
- get
- read
- init
- stop
- decimate
- labels
- sum
- grab
- print
- transfer
- release
- remove
- status
- swap
- param
- log
- error

darcmagic set

- darcmagic set NAME VALUE
    - VALUE gets exec'd – so can be simple python code
- darcmagic set NAME -file=FILE.fits
- darcmagic set -name=NAME -string=STRING
- darcmagic set -name=NAME -value=VALUE
- darcmagic set NAME VALUE COMMENT
- darcmagic set NAME VALUE -comment=COMMENT
- Other options include:
    - -swap=1/0 (default 1, do a buffer swap)
    - -check=1/0 (default 1, check the parameter for validity
    - -copy=1/0 (default 1, to copy newly active buffer into inactive buffer)

darcmagic get

- darcmagic get NAME
- darcmagic get -labels
- darcmagic get -log
- darcmagic get -version
- darcmagic get -decimation
- darcmagic get -name=NAME
- darcmagic get NAME -comment
- other options:
  - -print=1/0 (default 1, to print to terminal)
  - -file=FILENAME (to write to a text file)
  - -FITS=FILE.fits (to write to a FITS file)
  - -plot=0/1 (to plot the result)

darcmagic read

- darcmagic read rtcPxlBuf 10
- darcmagic read -s1=rtcPxlBuf -s2=rtcMirrorBuf -s3=rtcCentBuf -n=10
- darcmagic read -s1=rtcPxlBuf -o1=myPixelData.fits 1000
- other options
  - -sN=STREAMNAME
  - -oN=FILE.fits (filename to which to save this stream N to, defaults to STREAMNAME.fits)
  - -f=FRAME (the frame number at which recording should start – must be in the future)
  - -d=DECIMATION (the decimation value at which to record, 1 if not specified)
  - -n=NFRAMES (can be -1 for an infinite number of frames)
  - --save-on-fly (to stream data to disk, rather than to memory – this is the default for n<0 or n>10000)
  - -getstate=0/1 (default 0, if set, saves the RTC state with the data)
  - -print=0/1 (print data to screen)
  - -nosave=0/1 (don't save the data)
  - -plot=0/1 (plot the data as it arrives)

darcmagic init

- darcmagic init FILENAME.py
- darcmagic init -file=FILENAME.py
- darcmagic init FILENAME.fits
- darcmagic init FILENAME.py -remote
  - For files that live on the server, not locally
- Note – if the configuration file causes anything to be loaded from disk (e.g. a FITS file background map), this must exist on the server.
  - Can use darcmagic remote for this (see later)

darcmagic labels

- Gets a list of available parameters
- darcmagic labels
- darcmagic get -labels
- darcmagic labels -file=filename.txt -print=0

darcmagic decimate

- darcmagic decimate
- darcmagic get -decimation
- darcmagic decimate STREAM VALUE
- other options
  - remote=1/0 (set values on RTC)
  - local=1/0 (set values on local SHM buffer)

darcmagic sum

- darcmagic sum STREAM NFRAMES
- options
  - -s=STREAM
  - -n=NFRAMES
  - -d=DATATYPE (or n for as original data)
  - -a=0/1 (to average after summing)
  - -print=1/0
  - -file=FILE.fits
  - -plot=0/1

darcmagic grab

- darcmagic grab STREAM
- darcmagic grab rtcPxlBuf
- options
  - -latest=0/1 (if set, gets the latest frame, rather than waiting for a new frame)
  - -print=1/0
  - -tostring (convert data to ascii – e.g. rtcStatusBuf)
  - -file=FILE.fits
  - -plot

darcmagic stop

- darcmagic stop
  - Stops the RTC
- darcmagic stop -c
  - Also stops the control object

darcmagic print

- darcmagic print
- darcmagic print -file=FILE.txt -print=1/0

darcmagic transfer

- darcmagic transfer FILE
- options
  - -remote=REMOTEFILE
    - allows name change, i.e. copies local FILE to remote REMOTEFILE

darcmagic release

- Used in emergencies
  - Releases the CORBA lock
  - Shouldn't be required but may be useful during development

darcmagic param

- Subscribe to changes in the parameter buffer
  - Will be notified when a parameter changes
- options
  - -file=FILENAME
  - -print=1/0

darcmagic log

- Reads the RTC logs
  - control log and darc log
- Options:
  - -file=FILENAME
  - -darc=1/0 (subscribe to DARC log)
  - -control=1/0 (subscribe to control log)

darcmagic error

- Gets a list of current errors
- darcmagic error clear
  - clears all errors
  - Options:
    - -file=FILENAME.txt
    - -print=1/0
    - -log
      - Logs the errors (i.e. doesn't return)
- darcmagic error clear “ERRORSTRING”
  - clears a specific error

Other languages

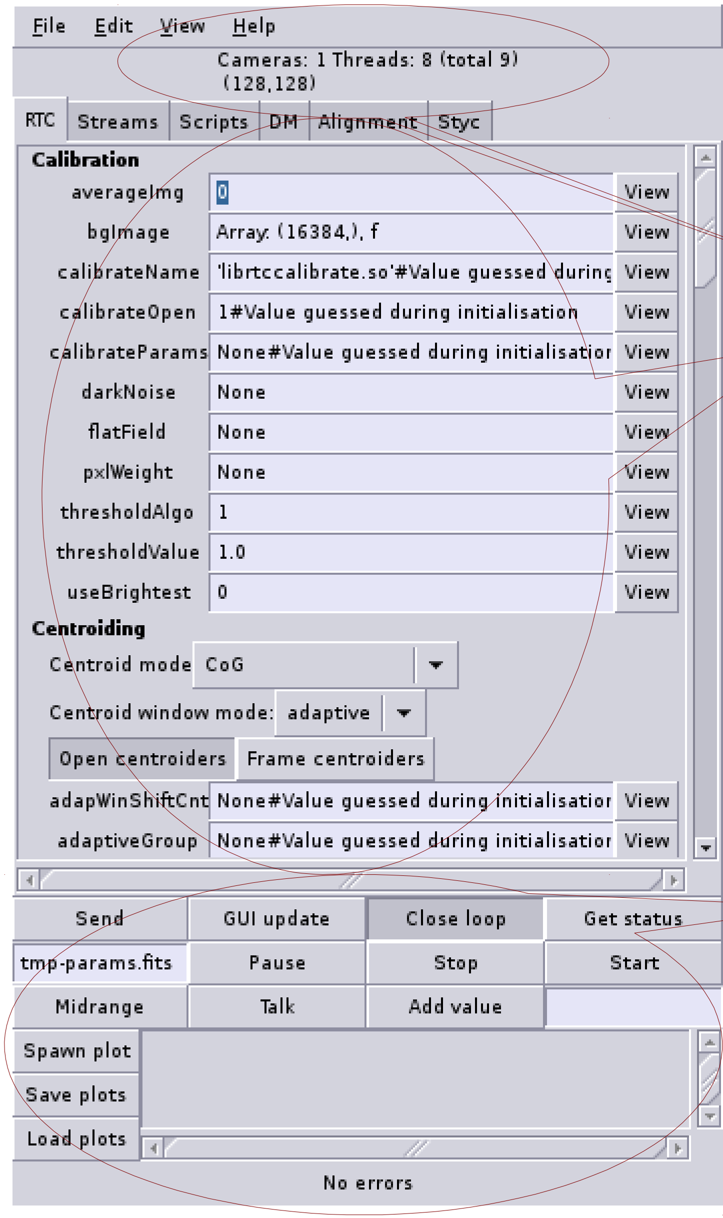
- Either:
  - implement CORBA
  - Or python bindings
  - Or use the command line tools
    - For CANARY this is what Yorick uses

Errors and logs

- darcmain writes errors to a telemetry buffer
  - /rtcErrorBuf
  - Clients can then read this
- To write an error from your own modules:
  - writeErrorVA(rtcErrorBuf,-1,iter,"Error string")
    - where rtcErrorBuf has been provided by the *Open() function
    - iter is the current iteration

The engineering GUI

- darcgui
- Used to control parameters and view data
  - control of telemetry too
- Basic functionality
- Users will also eventually need to implement their own tools...
  - eg. STYC for CANARY



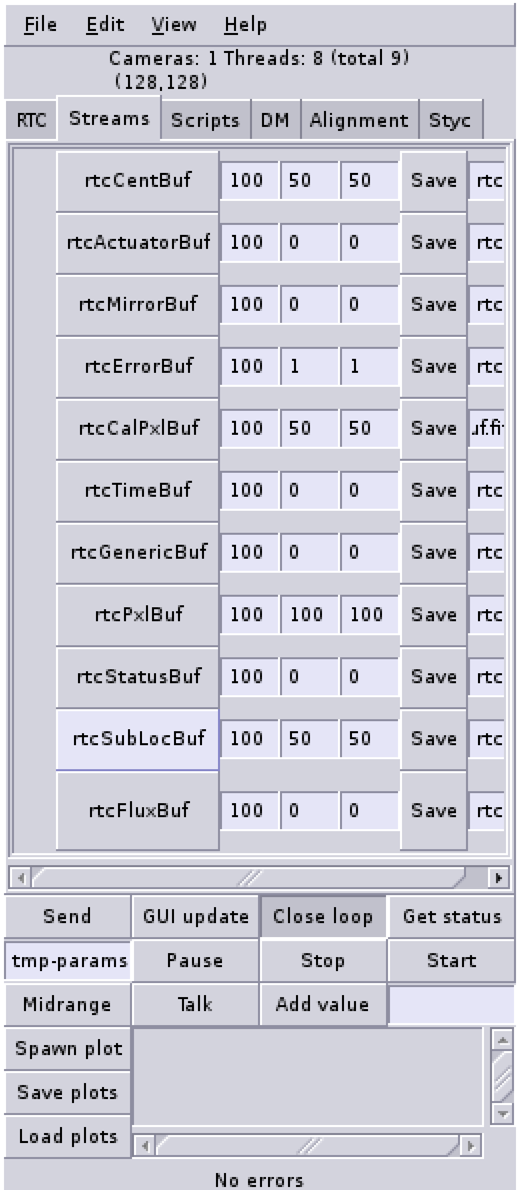
GUI parameters

- Basic info
- Parameters
  - name, value#comment
  - view button
  - Obtained from the RTC
  - User can update
    - Green if valid
    - Red if invalid python
    - Purple if invalid type/size
    - Black if same as in RTC
- Other stuff
  - Starting/stopping
  - Opening/closing loop
  - Status
  - Send/update
  - Plotting

Plotting

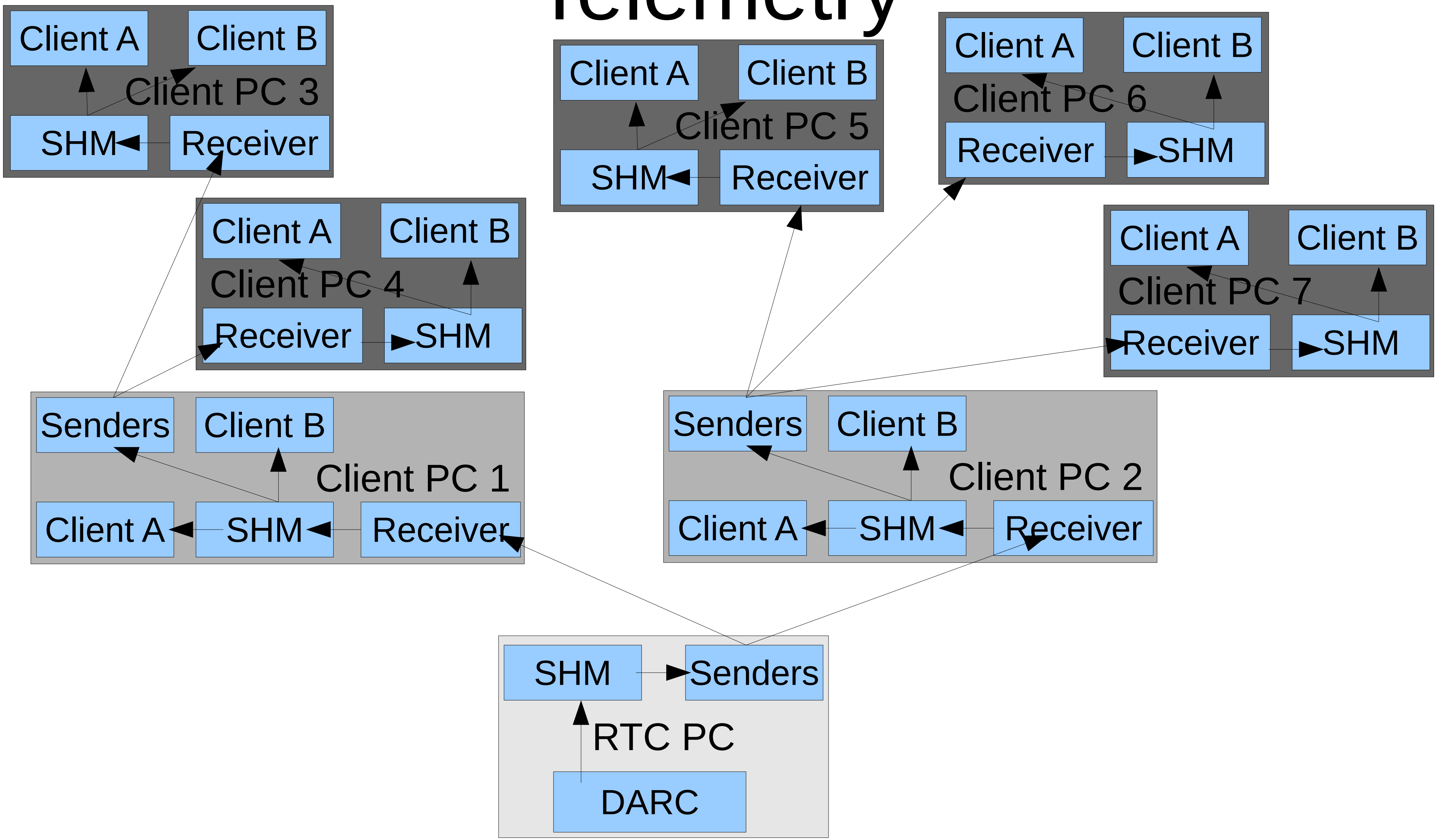
- Spawn plot
  - Starts a new plot process
  - User then selects what to plot
- Load plots
  - Starts plots based on a pre-written plot configuration file
- Save plots
  - Saves current plot configurations to a plot configuration file – which can later be loaded

GUI telemetry



- Turn on/off
- Grab (right click)
- Set RTC decimation
- Set local decimation
- Set GUI decimation (only for saving)
- Save data

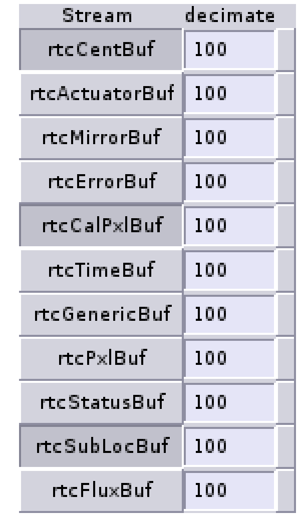
Telemetry

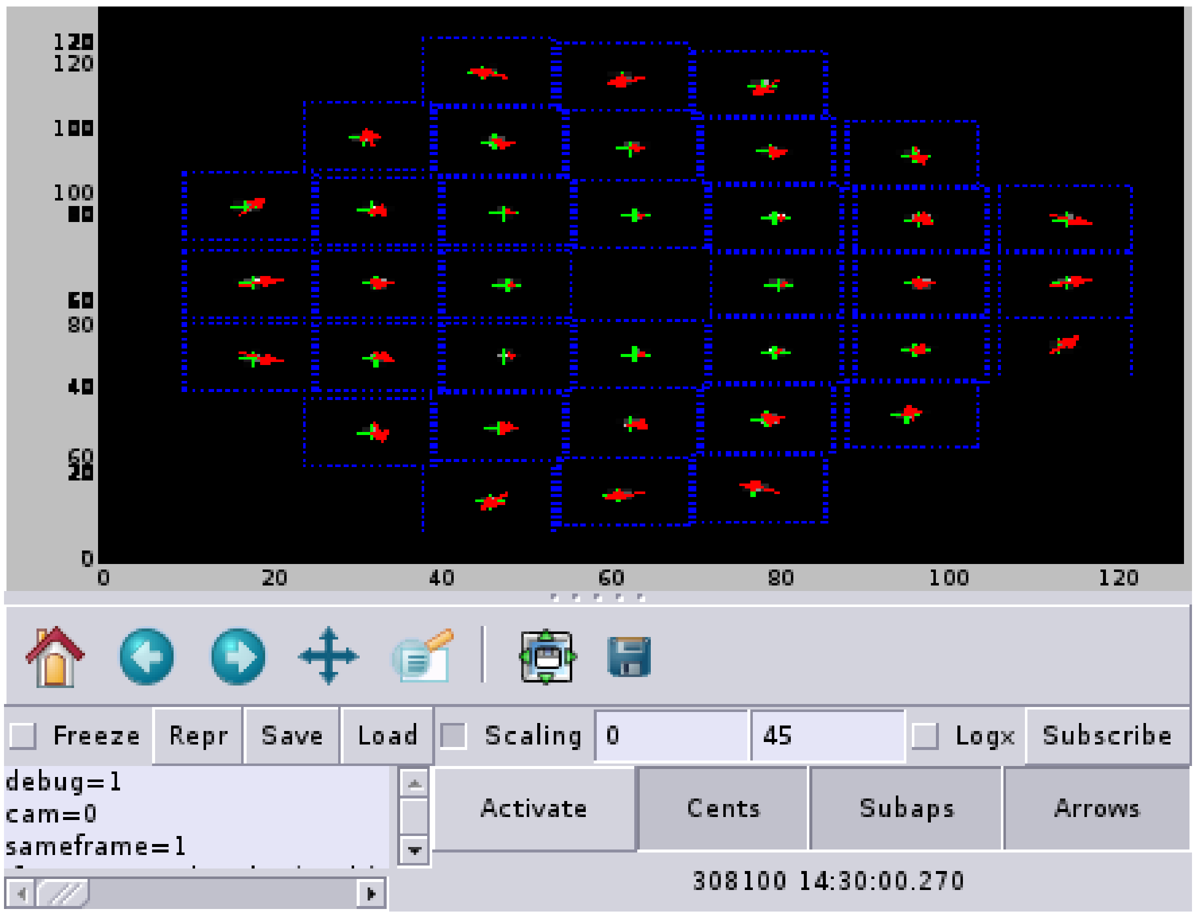

Client A
Client B
Client PC 3
SHM
Receiver
Client A
Client B
Client PC 5
SHM
Receiver
Client A
Client B
Client PC 6
Receiver
SHM
Client A
Client B
Client PC 4
Receiver
SHM
Client A
Client B
Client PC 7
Receiver
SHM
Senders
Client B
Client PC 1
Client A
SHM
Receiver
Senders
Client B
Client PC 2
Client A
SHM
Receiver
SHM
Senders
RTC PC
DARC

Real-time displays

- From the engineering GUI or...
- Command:
  - darcplot (lib/python/plot.py)
  - darcplot STREAM DECIMATION
  - Other options:
    - A plot configuration filename
    - -s=STREAM,STREAM,STREAM
    - -d=DECIMATION
    - -mPre-plot command
      - e.g. darcplot rtcPxlBuf 100 -mdata.shape=128,128
- Note, the plotter is generic
  - It receives a 1D array of whatever stream(s) subscribed too
  - The user can then configure it to display this as they wish

The plotter





- Multiple subscriptions
- Reformat however you like...
- Control local decimation rate
- This shows a calibrated image, the current sub-aperture locations and measured slopes
- A right click hides/shows the full image (removes the stuff at the bottom)
- Click “subscribe” to display currently available streams, and current subscriptions

the plot GUI

- A right click gives a menu
  - Can then subscribe/unsubscribe
  - Change how the data is displayed
  - e.g.

    data.shape=128,128

    data=data[30:40]

    dim=1

    axis=numpy.arange(128)
- Also shows the min/max values
- And gives some user configurable buttons

Plot configuration file example

- For complex displays, a plot configuration file can be used
- e.g. darcplot conf/plotcalimg1cam.xml
  - Note, here again, the plotter is generic – but the xml file has told it how to display the data.
- Now turn on adaptive windowing...
  - darcmagic set windowMode -string=adaptive

Asynchronous operation

- Using multi-rate cameras can give AO performance improvements
  - Slower frame rates for fainter guide stars
- To make use of this, we have to run DARC in asynchronous mode
  - This is done by running several instances of DARC
  - Lets have a go...

    darccontrol conf/configAsync.py

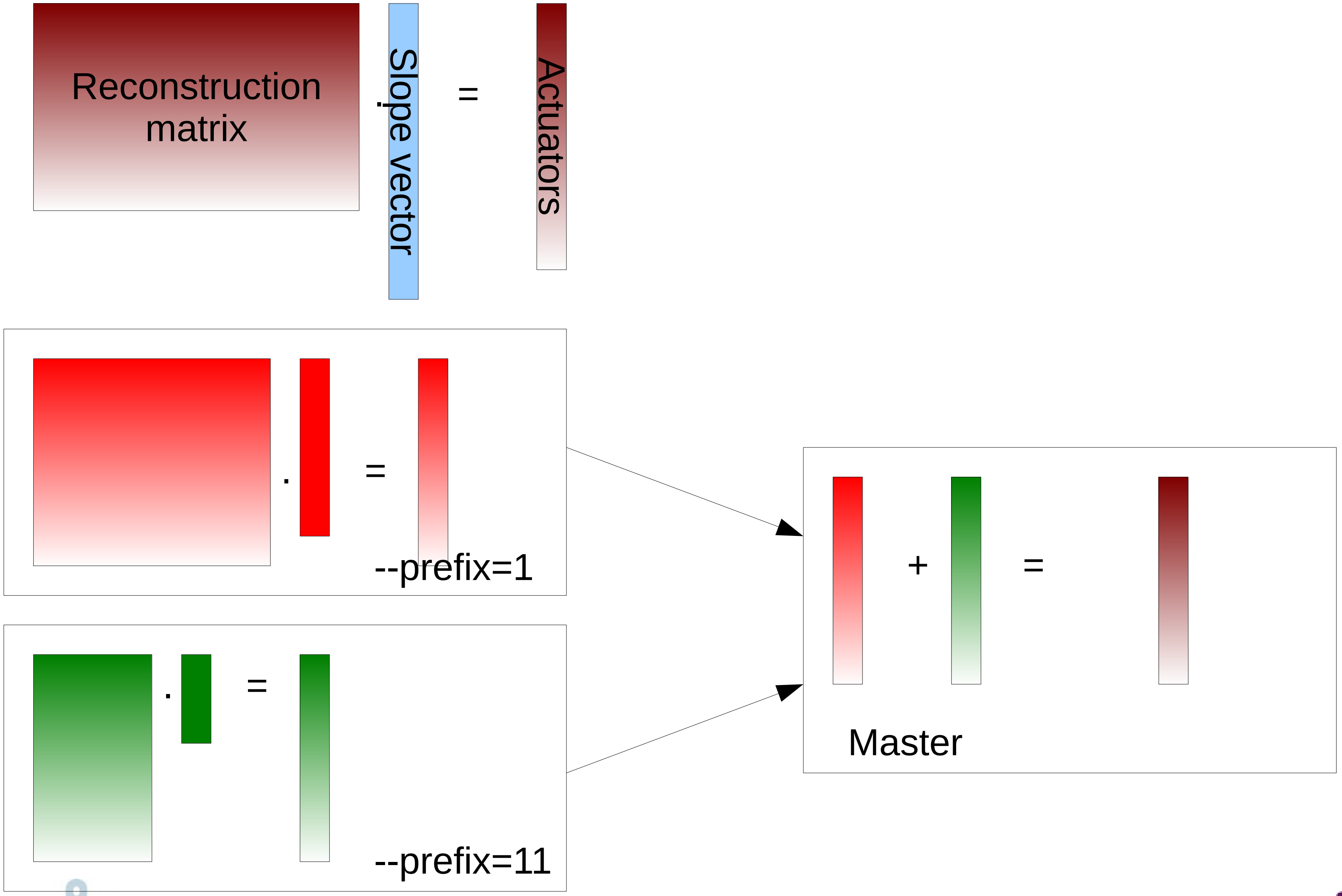
    darccontrol conf/configAsync.py --prefix=1

    darccontrol conf/configAsync.py --prefix=11
  - darcmagic status (--prefix=1, --prefix=11)
  - darcmagic sum rtcTimeBuf 100 -a (--prefix=1, --prefix=11)
    - computes average frame time

Asynchronous details

- How does this work?
- Multiple instances process their pixel data, and perform the reconstruction relevant for this WFS
  - in our example, prefix's 1 and 11
- The reconstructions are then sent to a “master” DARC instance
  - Which combines them and drives the mirror
  - The mirror is updated every time the master gets new commands
  - The master can be configured to group “slaves” together
    - so that update only occurs once all slaves have sent commands
- Master receives the partial commands by socket or shared memory
- But since it is just a reconstruction module, it can be easily changed

Async partial reconstruction



To note

- Async operation requires multiple instances of DARC
- Makes it slightly harder for the end user to interface
  - i.e. multiple pixel streams, rather than just one
  - multiple control objects
  - if this feature is likely to be an asset, should be designed into the end user software from the start

Distributed processing

- If you have a CPU cluster:
  - Can use the asynchronous operation principals to distribute DARC over available PCs
  - Either:
    - Split on a sub-aperture basis
      - e.g. all cameras → partial reconstruction on separate PCs
        - Recombined on another
      - Or even split up pixel data using a cable splitter
        - e.g. Half pixels to one PC, half to another
          - Then partial reconstruction on each, followed by recombination
    - Or split on a processing chain basis
      - Calibration on one (or more) PC
      - Slope processing on another (or >1)
      - Reconstruction on another (or >1)
      - Mirror control on another (or >1)
        - You just need to develop modules to send and receive data at appropriate points
    - Or a combination of both
  - So – highly scalable

The buffer module

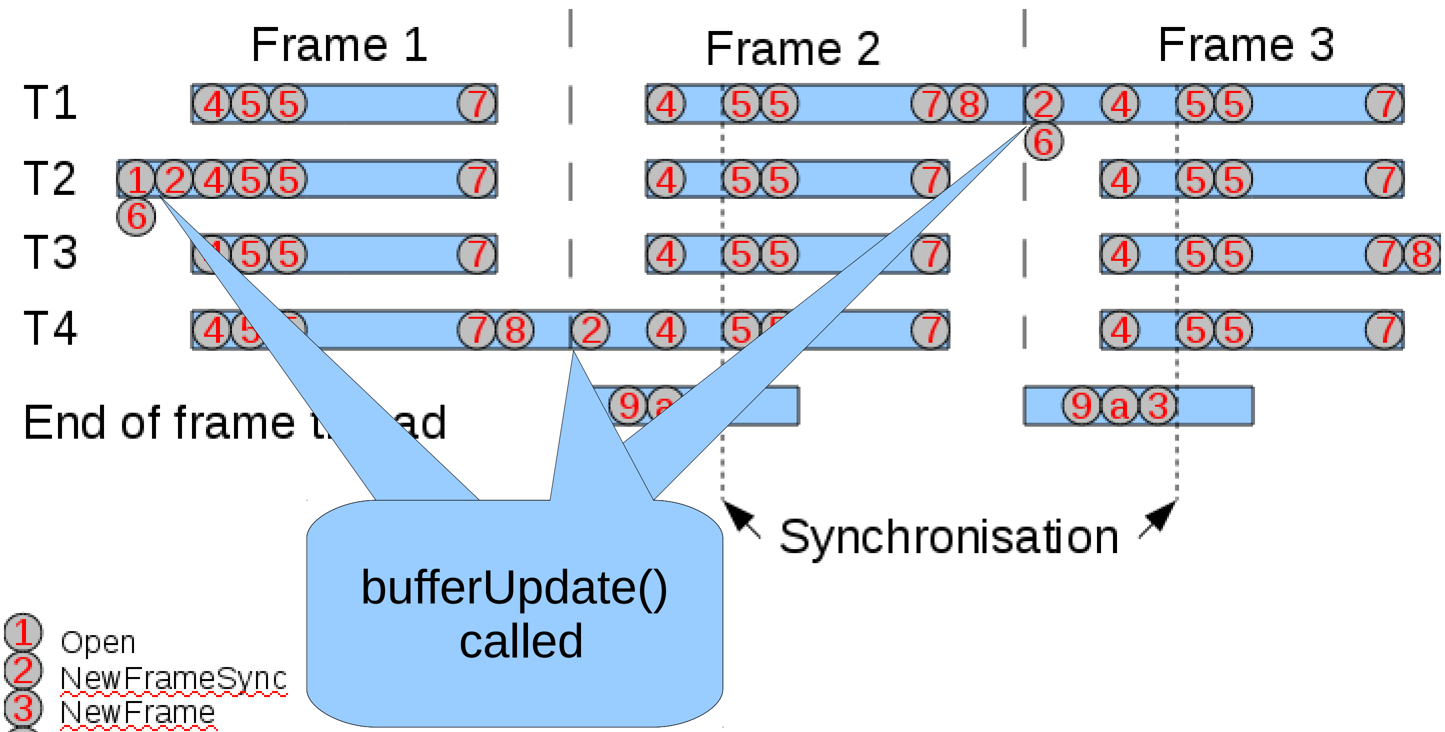
- Changing a parameter can be slow
  - network delays
  - CORBA delays
  - python delays
- At best it may take several frames to update, before the next parameter can be changed
- But what if you want a parameter to change on a frame-by-frame basis?
  - (note – for DM offsets, can just send a 2D array – e.g. when poking)
  - But for anything else, need to use the buffer module
- Currently available: src/rtcbuffer.c
  - primarily as an example, but does have a use for pre-known parameters
- To be able to stream parameters into DARC (e.g. over TCP/sFPDP/infiniBand etc) need to implement your own

rtcbuffer.h

```
int bufferClose(void **bufferHandle);

int bufferNewParam(void *bufferHandle,paramBuf *pbuf,unsigned int frameno,
            arrayStruct *arr,paramBuf *inactive);

int bufferOpen(char *name,int n,int *args,paramBuf *pbuf,circBuf *rtcErrorBuf,
         char *prefix,arrayStruct *arr,void **handle,int nthreads,
         unsigned int frameno,unsigned int** bufferframeno,
         int *bufferframenosize,paramBuf *inactive);

int bufferUpdate(void *bufferHandle);
```

- bufferNewParam – called when the parameter buffer is updated in the standard way
- bufferUpdate – called before any newFrameSync functions
  - This is called every iteration and can be used to update the active (or inactive) parameter buffers
  - e.g. with new data received from a socket
  - It can also trigger a parameter buffer swap

bufferUpdate position



1. Open
2. NewFrameSync
3. NewFrame
4. StartFrame
5. camWaitPixels (or calibrateNewSubap or slopeCalcSlope or reconNewSlopes)
6. NewParam
7. EndFrame
8. FrameFinishedSync
9. FrameFinished
a. Complete

Current buffer module src/rtcbuffer.c

- librtcbuffer.so
- Requires a bufferSeq parameter in the parameter buffer
- If this is present, it is used as a sequencer for the bufferUpdate() function
  - e.g. allows changing of reference slopes each frame, for example for modulation
    - But not in response to feedback

Figure sensing

- Open-loop AO systems with non-linear DMs require a figure sensor
  - We know what shape we want the mirror to take
    - Using on-sky WFSs and wavefront reconstruction
    - The “desired shape” (actuators)
  - A figure sensing WFS images the DM with a bench laser source
    - Gets the actual shape of the DM
    - This is adjusted until actual shape matches “desired shape”
    - FS WFS usually operates at higher frame rate than on-sky WFS
      - Allows several iterations to adjust the DM shape
  - But – how do we get the “desired shape” into the FS DARC?
    - Using the figure sensing module

The figure sensing module

```
int figureOpen(char *name,int n,int *args,paramBuf *pbuf,circBuf *rtcErrorBuf,
            char *prefix,arrayStruct *arr,void **handle,int nthreads,
            unsigned int frameno,unsigned int **figureframeno,
            int *figureframenoSize,int totCents,int nacts,
            pthread_mutex_t m,pthread_cond_t cond,float **actsRequired);

int figureClose(void **figureHandle);

int figureNewParam(void *figureHandle,paramBuf *pbuf,unsigned int frameno,
               arrayStruct *arr);
```

- 3 functions only
- The figureOpen function should start a new thread:
  - This thread accepts the “desired shape” from the on-sky DARC
  - Once a new shape is received, locks the mutex (provided to figureOpen() call)
  - Writes required shape to actsRequired
  - Signals the condition variable
  - unlocks the mutex
- The new mirror demands are then calculated and sent immediately – at the on-sky WFS frame rate
- Also – the main loop of this FS DARC:
  - Reads the WFS, reconstructs, factors in the desired shape, and sends
    - At the FS WFS frame rate

Figure sensing examples

- For CANARY we used
  - src/figureSL240SC.c
    - A version for receiving actuators over sFPDP
  - src/figureSL240SCPassThrough.c
    - A version that received actuators, and placed them straight onto the mirror
      - dumb mode – actuators unmodified

Note: As a camera tool...

- At Durham, we're also using DARC as a generic camera tool
  - Easy to write a camera module for new cameras
    - Provided they have Linux drivers...
  - We then have a standard, reliable way of getting pixel data, standardised around the lab...
  - Saves effort

Kernels

- Stock kernels may not be optimised for DARC
  - Server kernels can have high latency
  - Different kernel numbers offer different performance
- Real-time kernels will reduce jitter
  - May also improve average latency (but may not)
  - Usually available from Distro repository

Notes

- DARC isn't a finished product
  - Continued future development
    - Large array support (>4GB control matrices)
    - Additional algorithms
    - etc
- Can you use it?
  - Do you require any modifications?
- Have we missed anything?
- Bugs?
- Anything else?