

DDX

A Distributed Software Architecture for Robotic Systems

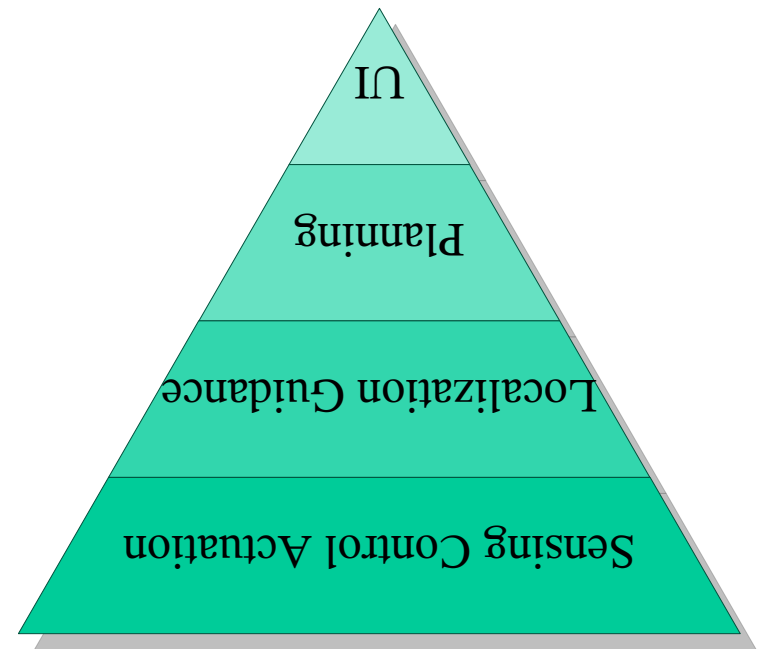
Peter Corke, Pavan Sikka, Jonathan Roberts and Elliot Duff

Presented by: Elliot Duff

ACRA 2004

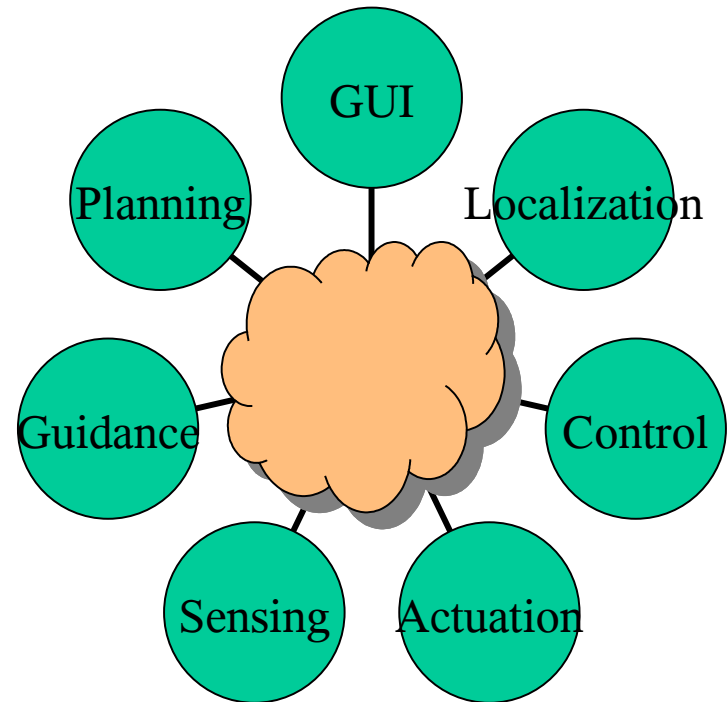
Monolithic Robotic Systems

- Advantages
 - Deterministic
 - Real time performance
- Disadvantages
 - Does not scale well
 - Tight coupling between modules
 - Inflexible



Distributed Architecture

- Advantages
 - Scales very well – just add CPU's
 - Flexible – just replace modules
- Disadvantages
 - Communication
 - Performance
 - Implementation ?



1. OOM - Object Oriented Middleware

- DCOM - Distributed Component Object Model (COM)
- CORBA - Common Object Request Broker Architecture (ORB)
- RMI – Remote Method Invocation (JVM)
- Web Services (XML-RPC, SOAP, etc)

2. MOM- Message Oriented Middleware (data-centric)

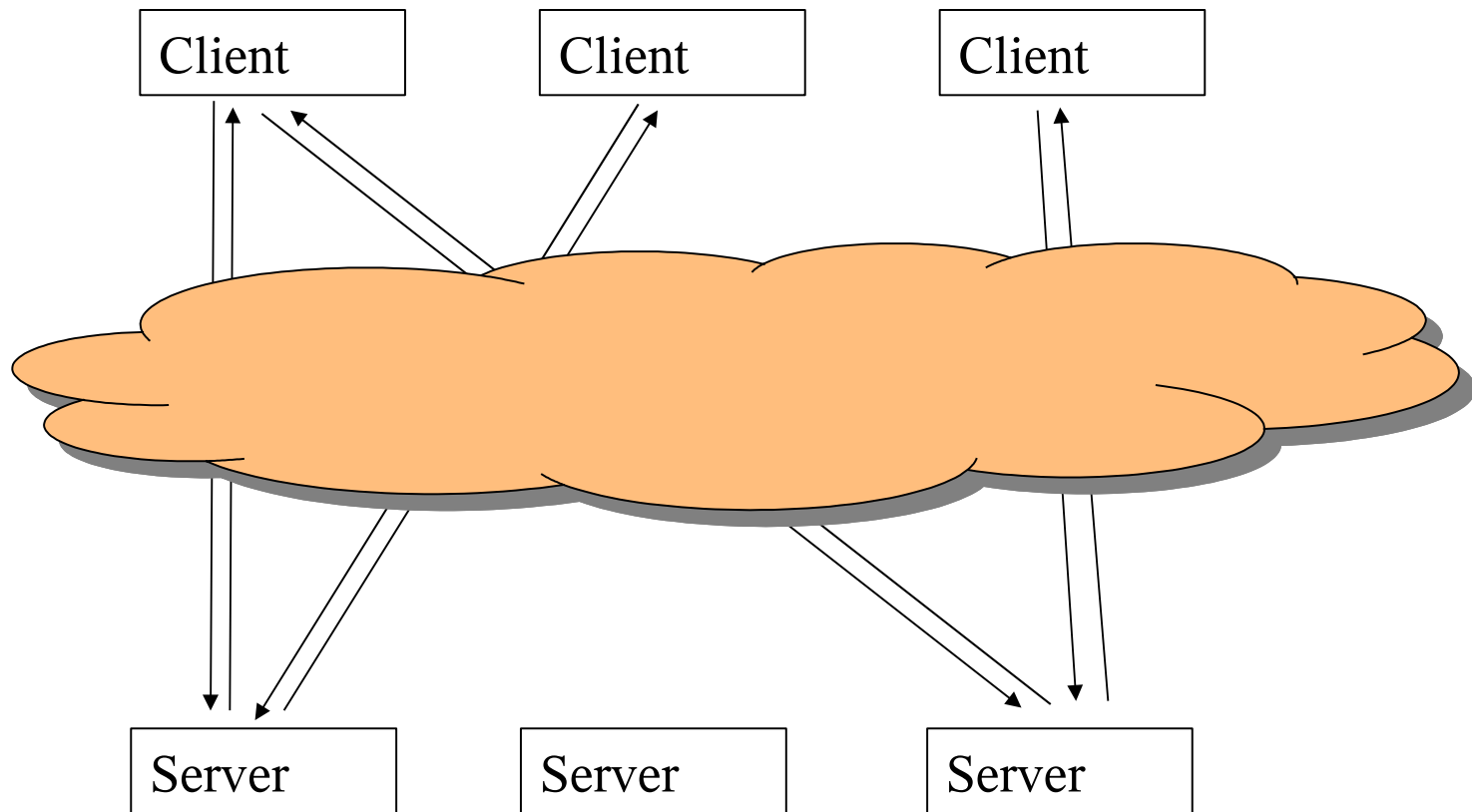
- JMS – Java Messaging Service
- MSMQ – Microsoft Messaging Queue

3. EOM - Event Oriented Middleware (real-time)

- CORBA Event Services – TAO Real-time event service
- DDS Data Distributed Services – NDDS Real Time Innovations

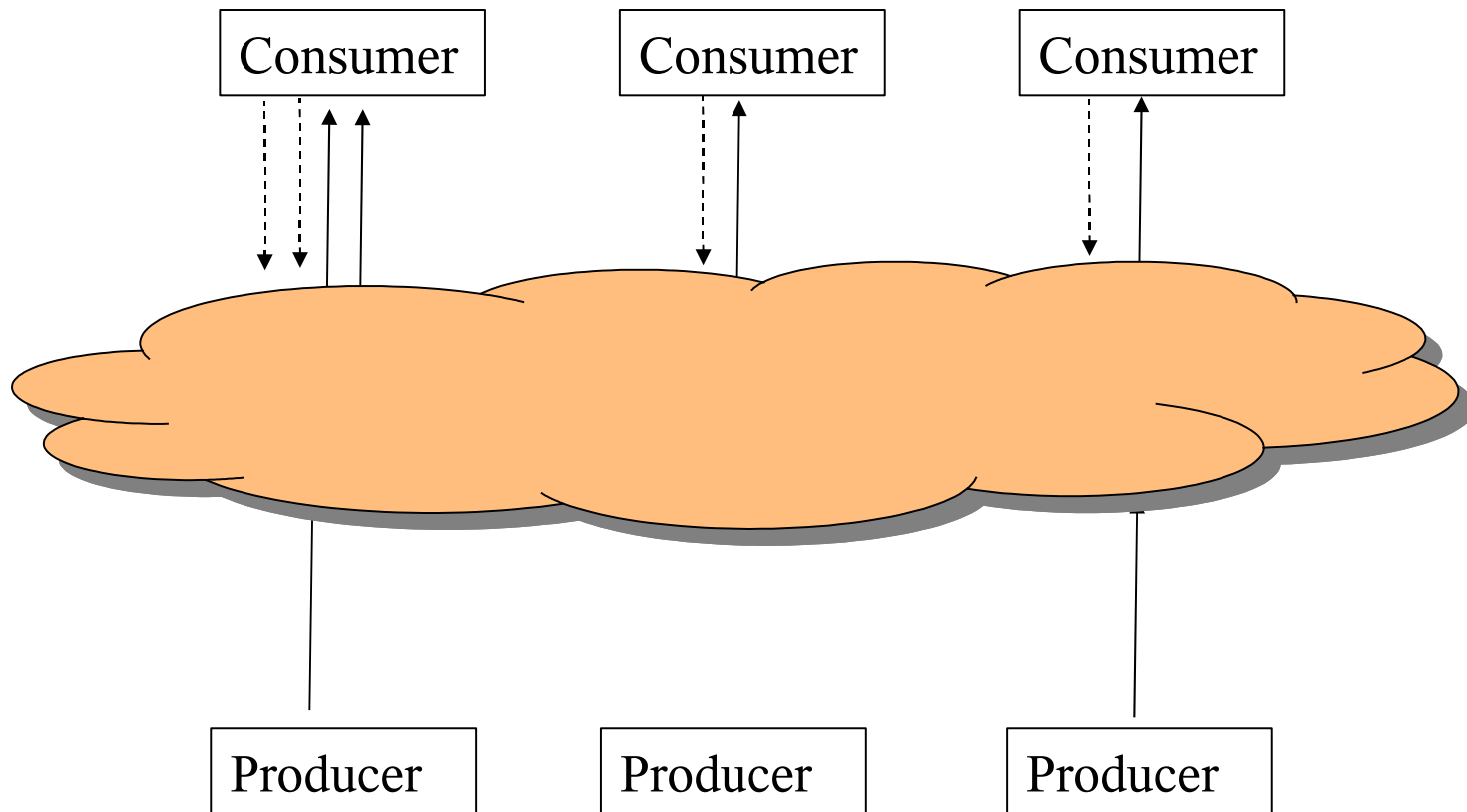
Client / Server

www.ict.csiro.au



Producer / Consumer

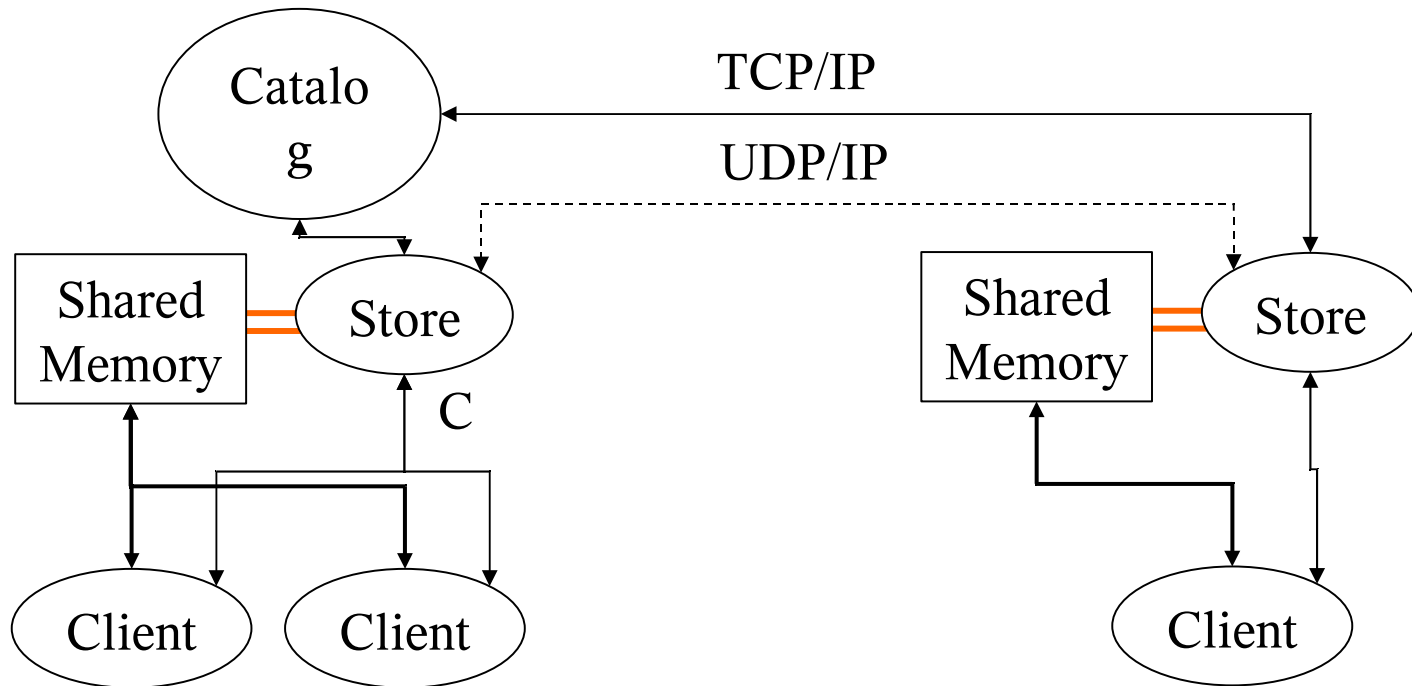
www.ict.csiro.au



- Our requirements:
 - Safe and reliable mechanism for shared data
 - Provide real-time control
 - Scalable and flexible
 - Low overheads
 - Time-stamping of all data
 - Ability to log data to disk
- CORBA
 - Based upon network protocol
 - Pluggable protocols to support shared memory
- Alternative
 - System that uses shared memory
 - but supports network when needed

DDX components

www.ict.csiro.au



Client API – Simple data

```
DDX_STORE_ID *storeID;  
DDX_STORE_ITEM *itemPtr;  
int i = 10;  
  
ddx_client_init(0);  
storeID = ddx_store_open(NULL,0,2);  
itemPtr = ddx_store_lookup_item(storeID, "frame", "int", sizeof(int));  
ddx_store_write(itemPtr, &i, NULL);  
ddx_store_done_item(itemPtr);  
ddx_store_close(storeID);  
ddx_client_done();
```

```
ddx_client_init(0);  
storeID = ddx_store_open(NULL,0,2);  
itemPtr = ddx_store_lookup_item(storeID, "frame", NULL, 0);  
ddx_store_read(itemPtr, &i, NULL, 0.0, 0);  
printf("%d\n", i); }  
ddx_store_done_item(itemPtr);  
ddx_store_close(storeID);  
ddx_client_done();
```

What if ?

- You don't know the type of data?

```
itemPtr = ddx_store_lookup_item(storeID, "frame", NULL, 0);  
printf("%s\n", ddx_store_item_get_definition(itemPtr));
```

- You do, and want to check ?

```
itemPtr = ddx_store_lookup_item(storeID, "frame", "int", sizeof(int));
```

- You want the time that it was created?

```
rtxTime ts;  
ddx_store_read(itemPtr, &i, &ts, 0.0, 0);
```

- You want synchronous read?

```
ddx_store_read(itemPtr, &i, &ts, 1.0, 1);
```

- You want to read directly (without memcpy) ?

```
P = ddx_store_var_pointer(ItemPtr);  
ddx_store_read_direct(storeID, &ts, 1.0, 1);
```

More complex data types

1. Define simple data structure:

```
typedef struct { int valid, double value; } Encoder;  
Encoder enc;
```

2. Register data type:

```
ddx_store_register(storeID, "struct { int valid, double value; } Encoder;" );
```

3. Lookup item:

```
itemPtr = ddx_store_lookup_item(storeID, "encoder", "Encoder", sizeof(Encoder));
```

4. Write data to store:

```
ddx_store_write(itemPtr, &enc, NULL);
```

1. Lookup item:

```
itemPtr = ddx_store_lookup_item(storeID, "encoder", NULL, 0);
```

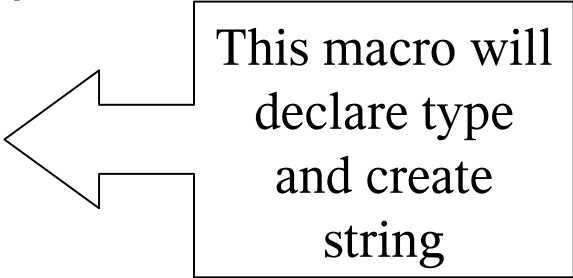
2. Read data from store:

```
ddx_store_read(itemPtr, &enc, 0.0, 0);
```

Let macros to do all the work

- Declare store data type

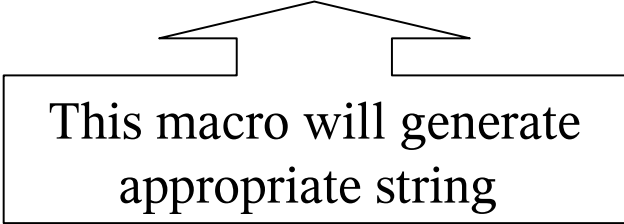
```
DDX_STORE_TYPE(Encoder,  
    struct {  
        int valid;  
        double value;  
    } );
```



This macro will
declare type
and create
string

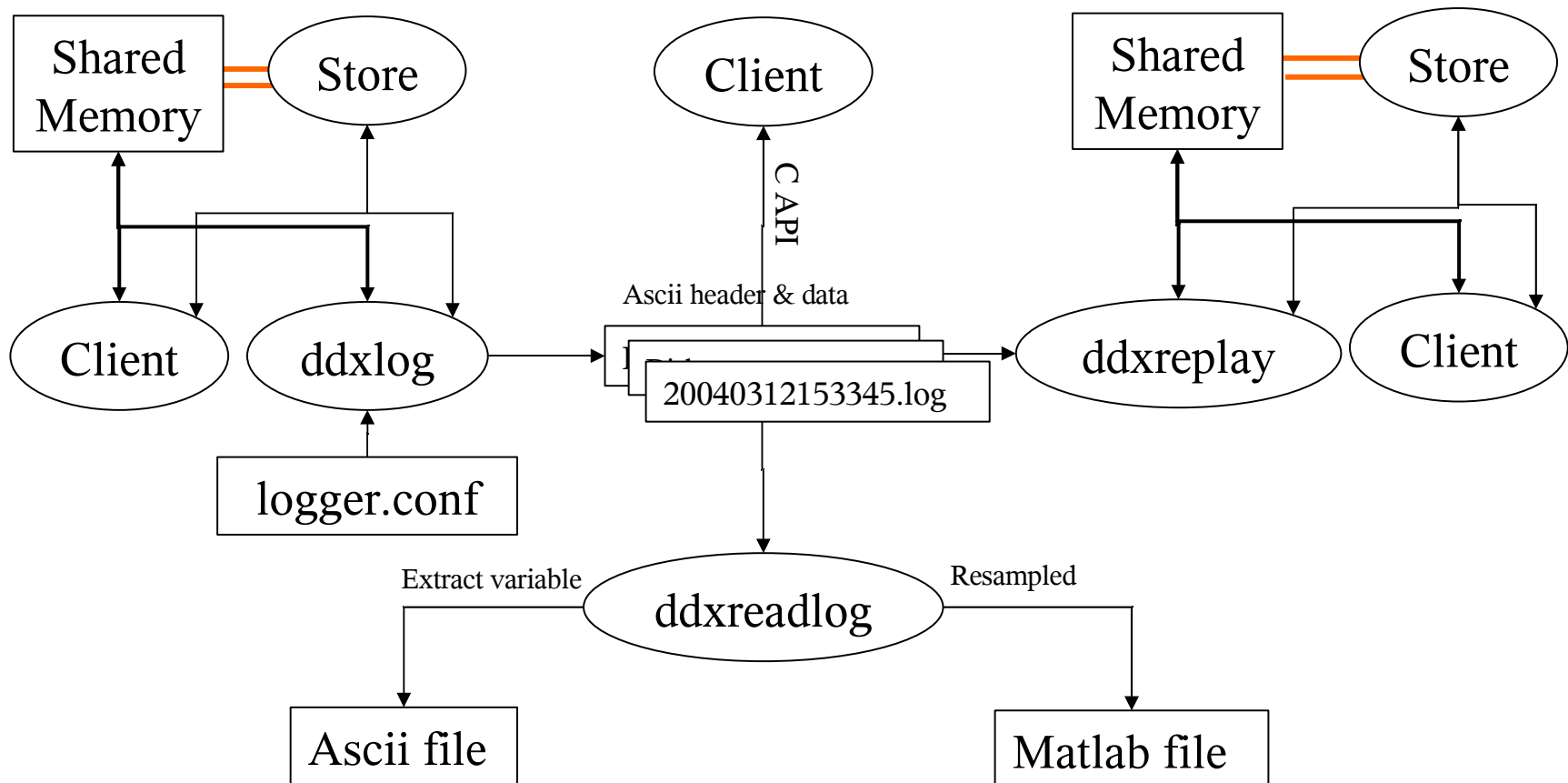
- Register data type

```
DDX_STORE_REGISTER_TYPE(storeID, Encoder);
```



This macro will generate
appropriate string

Data Logging



Interactive text console

www.ict.csiro.au

➤ `ddxsh -s emma-ph`

store@emma-h: ls

imu

Pls

test

store@emma-ph: ls -l

imu [size = 220] [count = 423]

pls [size 1028] [count = 1133]

test [size = 8] [count = 0]

store@emma-ph: ls test

struct {

int x;

int y;

};

store@emma-ph: print test

0.0 [0]

Application: Autonomous Tractor

www.ict.csiro.au

Software Modules

planning

GUI

target-tracking

localization

road-following

guidance

optical flow

avoidance

Vision (stereo)

tractor

Vision (omni)

joystick

Vision (fisheye)

gps

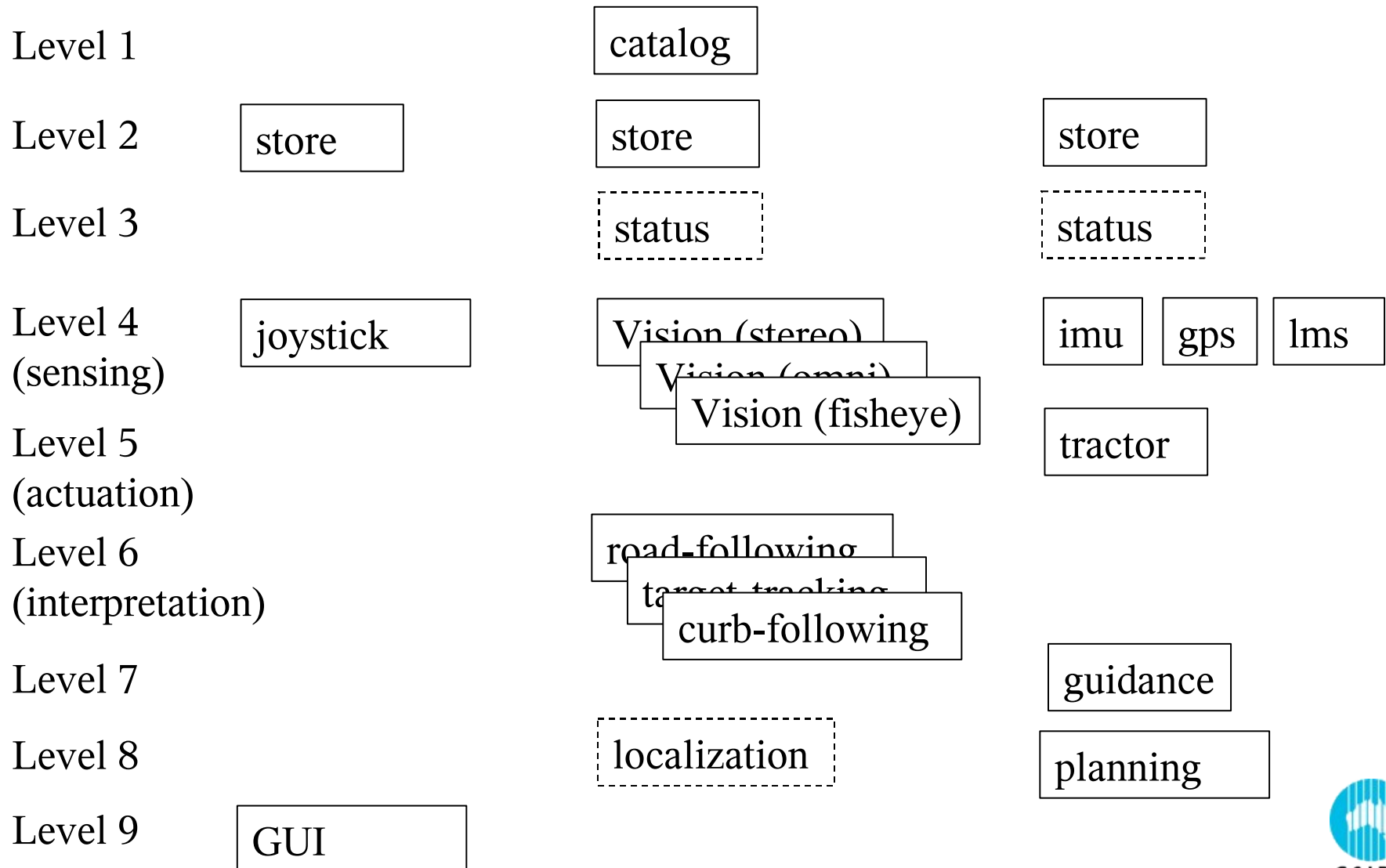
lms

imu



Managing programs and run levels

www.ict.csiro.au



> catalog -c tractor_test1.launch

```
1 catalog launch-aware 10 headroom:catalog
2 store1 launch-aware 10 headroom:store
2 store2 launch-aware 10 tractor:store c headroom-f
2 store launch-aware 10 store
4 pls launch-aware 10 tractor:pls -d ttyS8 -store
4 gps launch-aware 10 tractor:gps -d ttyS7:9600 -type sr530 -store
4 imu launch-aware 10 tractor:imu -d ttyS6 -store
4 omni launch-aware 10 headroom:ddxvideo1394 -number 2 -format 7 -mode 5
4 fish launch-aware 10 headroom:ddxvideo1394 -number 0 -format 0 -mode 3
4 joy launch-aware 10 ddxjoystick -store
5 tractor launch-aware 10 tractor:tractor -c tractor.conf
6 curb launch-aware 10 headroom:curb -name omni
6 road launch-aware 10 headroom:road -name omni
6 tracking launch-aware 10 headroom:tracking -name fish
7 guidance launch-aware 10 tractor:snake -name vlaser
8 planning launch-aware 10 tractor:../bin/tractor-test
9 gui laubch-aware 10 ../bin/dashboard
```

- Deployed in many autonomous systems
 - Tractor / Submarine
 - Helicopter / AVS
 - Dragline / LHD
- Middleware is data-centric
 - Flow control is difficult
 - Use state variables
- Uses UDP communication
 - Restricted to 8K data
 - Problem for vision

- Language bindings for
 - Java, Python and Matlab
- Use Player / Stage ?
- Open Source ?

Questions ?