cosylab
CONTROL SYSTEM LABORATORY

Integration of ACS and LabVIEW

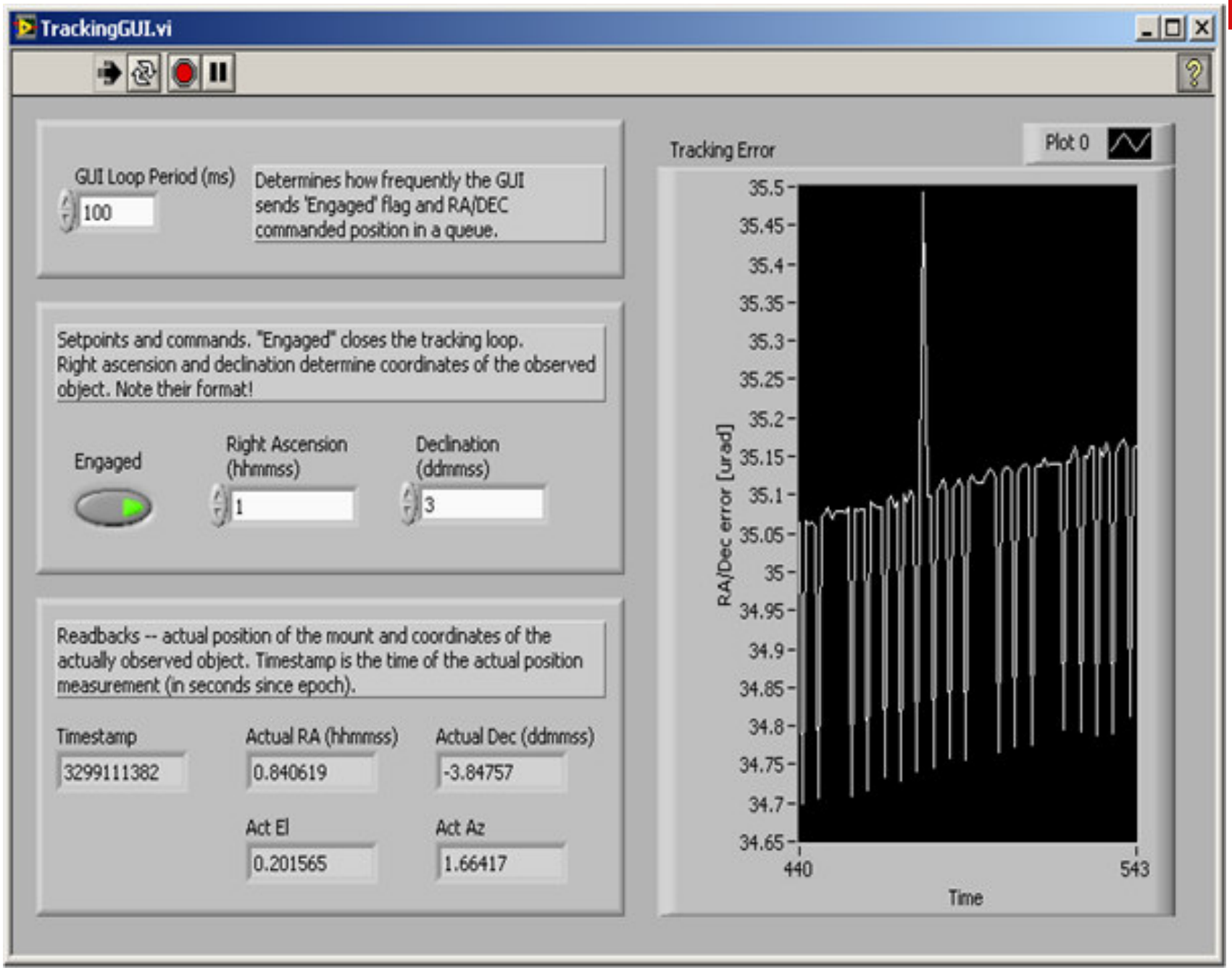
Klemen Žagar
klemen.zagar@cosylab.com

ACS

- ALMA Common Software
- “Integration layer”, focusing on manageability and maintainability of complex distributed control systems
- Offers infrastructural services:
  - Distributed logging
  - Publish/subscribe (message-oriented) middleware
  - Component-based software engineering
  - Lifecycle management
  - Infrastructure monitoring
  - Alarms
  - Configuration database
  - Bulk data transfer
- Based on CORBA, supporting C++, Java and Python
- Development started in 2000
  - now mature in its 7th release

LabVIEW

- National Instruments' toolkit for development of:
  - Control loops
  - Front-ends
- Easy to learn
- Short development cycles
- Graphical/visual development:
  - GUI composition
  - "Graphical programming"
- Large array of supported hardware
- Appealing widgets
- Rich library for data analysis
- Development of FPGA circuits
  - Without detailed hardware design knowledge (e.g., VHDL, routing, …)


TrackingGUI.vi
GUI Loop Period (ms)
100
Determines how frequently the GUI sends 'Engaged' flag and RA/DEC commanded position in a queue.
Setpoints and commands. "Engaged" closes the tracking loop. Right ascension and declination determine coordinates of the observed object. Note their format!
Engaged
Right Ascension (hhmmss)
1
Declination (ddmmss)
3
Readbacks -- actual position of the mount and coordinates of the actually observed object. Timestamp is the time of the actual position measurement (in seconds since epoch).
Timestamp
3299111382
Actual RA (hhmmss)
0.840619
Actual Dec (ddmmss)
-3.84757
Act El
0.201565
Act Az
1.66417
Tracking Error
Plot 0
RA/Dec error [urad]
35.5
35.45
35.4
35.35
35.3
35.25
35.2
35.15
35.1
35.05
35
34.95
34.9
34.85
34.8
34.75
34.7
34.65
440
543
Time

ACS + LabVIEW

- In LabVIEW, one would develop:
  - Simple control loops (say ~10 processing steps, no cutting-edge performance requirements)
  - Simple GUIs (e.g., static engineering/expert screens)

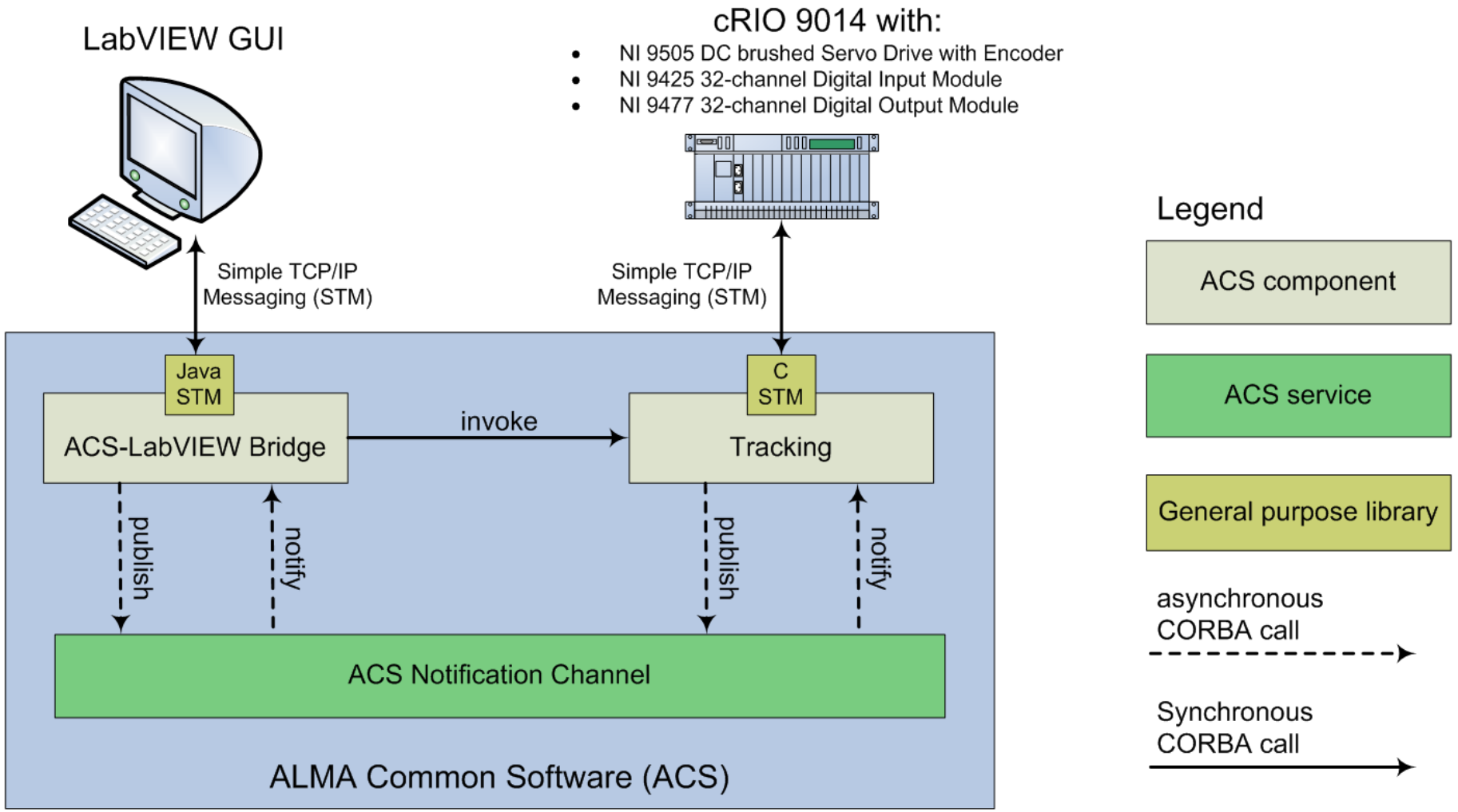
- In LabVIEW, one would rather not:
  - Build a complex system (many processing steps) – graphical programs tend to become difficult to maintain
  - Build a highly-distributed system (beyond a few nodes)
  - Dynamic GUIs
  - Squeeze the last bit out of the hardware (LabVIEW execution environment takes its toll)

- Were it not nice if we had the best of both worlds?

Approaches

- Call Library Node
  - ACS client code integrated into LabVIEW as a shared library
  - Platform dependent
  - Tricky to encapsulate ACS client in a shared library for Windows that wouldn't clash with LabVIEW run-time
- TCP/IP communication
  - ACS processes separate
  - LabVIEW's TCP communication blocks used to communicate with them
- LabVIEW shared variables and data sockets
  - Not equally supported on all platforms
  - Lack of control over the protocol

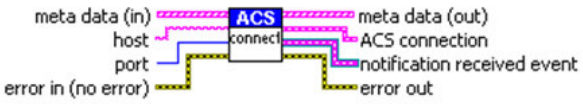
Architecture of the TCP/IP communication approach



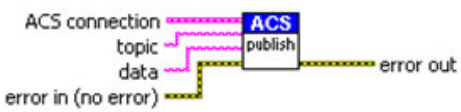
The LabVIEW API

- Simple TCP/IP API is provided by National Instruments (DevZone)
- We built an abstraction atop of it for working with ACS

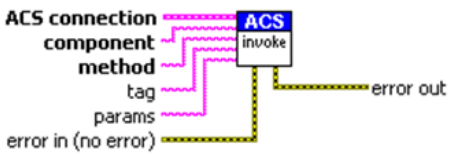
ACS Connect.vi



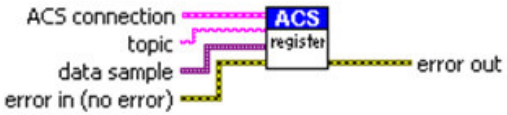
ACS Publish.vi



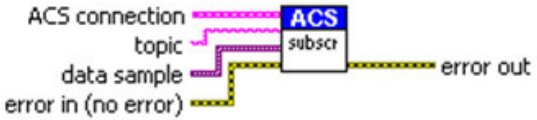
ACS Invoke.vi



ACS Register Cluster.vi



ACS Subscribe.vi



ACS Event Dispatcher.vi

The C API

```c
#include "lvstm.h"

// Connect to a LabVIEW process
lvstm_conn_t* c = lvstm_connect("host", port, read_callback, NULL, NULL);

// Prepare a message to send.
lvstm_message_t* msg = lvstm_create_message("Engaged");
lvstm_write_bool(msg, engaged);
// Send the message.
lvstm_send_message(c, msg);
lvstm_close_message(msg);

// Disconnect from LabVIEW
lvstm_disconnect(c);

// Called whenever LabVIEW sends a message
void read_callback(
  void* tag,
  lvstm_conn_t* conn,
  lvstm_message_t* msg
) {
    lvstm_read_double(msg, &act_az);
    lvstm_read_double(msg, &act_el);
    // ...
}
```

The Java API

```
// Initialization
lvstm = new Lvstm("ACS-LabView Bridge", new MyListener(), THREADS_NUM, true);
lvstm.start(true, socket);

// Termination
lvstm.stop();

/* Implementation of listener of LabVIEW-related events. */
public class MyListener implements ILvstmListener {
    /* LabVIEW has sent a message. */
    public void messageReceived(LvstmConnection connection, LvstmMessage message) {
        if (message.getMetaId() == subscribeMeta) {
            String topic = message.getStringIntLen();

            // Send a message back to LabVIEW
            connection.sendMessage(/* ... */);
        }
        // ...
    }

    /* LabVIEW has connected. */
    public void connected(LvstmConnection connection) {
        // ...
    }
}
```

Conclusion

- Applicable for integration of LabVIEW to anything
  - Pure C and Java API implementations
- Loosely-coupled approach:
  - Non-LabVIEW GUI talking to a LabVIEW control loop, and vice-versa (decision can be taken at run-time)
- Requires additional network “hops”, but sufficient for majority of applications
  - ~10Hz update rate, ~1kB payload/message
- Scalable: additional clients don’t put load on the LabVIEW control loop
- Optimized on-the-wire protocol
  - Data acquisition from LabVIEW can be transferred to a C/C++/Java process at high transfer rates
- Lesson learned:
  - Separate LabVIEW code in several concurrent control loops (GUI, communication, processing)
  - Use message queues to transfer data between them
  - Allows replacement (just run different sets of VIs)

Thank You for Your Attention