Company Overview

- KEC is a systems integrations company that specializes in National Instruments LabVIEW systems software development and hardware integration with more than 10 years of experience.
- Area of Expertise:
  - NI LabVIEW application/system software development
    - Ranging from single-tasked applications to multi-device multi-network system software design/development
    - Integration with NI hardware as well as 3rd party devices
  - Consultation services for hardware selection and LabVIEW software design
    - Working alongside customers’ engineering team to solve their challenges
    - Provide solutions to customers that caters to and adhere to their processes and requirements
Session Goals

- Introduce common & scalable communication architectures
- Understand what challenges each communication architecture solves
- Identify which protocols are ideally suited for different architectures
- Example application architectures
Why is Scalability Important?

(Client, Host, etc.)  (Node, Target, etc.)
Why is Scalability Important?
Why is Scalability Important?
Goal of Scalability

Minimize the effort and risk associated with adding “just one more”
Strategies for Scalability

- Use flexible communication architectures that require minimal foreknowledge of nodes in the system

- Use communication protocols that are optimized for the type of data being sent & received

- Design the system so data communications has little impact on the system performance and critical operations
Types of Data

Tag  
Current value data

Message/Command  
Intermittent data, reliable delivery

Stream  
Continuous flow, high throughput
## Protocols / Mechanisms

<table>
<thead>
<tr>
<th>Tag</th>
<th>Message</th>
<th>Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>• UDP</td>
<td>• TCP</td>
<td>• TCP</td>
</tr>
<tr>
<td>• PSP</td>
<td>• Network Streams</td>
<td>• Network Streams</td>
</tr>
<tr>
<td>• Shared Variables</td>
<td>• Web Services (REST)</td>
<td>• File Transfer</td>
</tr>
<tr>
<td></td>
<td>• Others (DDS)</td>
<td></td>
</tr>
</tbody>
</table>
Single Node Architecture

- **Communication** – 1 Client : 1 Node
- **Focus on Scalability** – Low / None
- Functionality is typically primary concern
- **Applications**
  - Simple control & monitoring
  - Prototyping
- **Potential Concerns**
  - Easy replacement of either client or node
System Replication
Multi-Node Architecture

- **Communication** – 1 Client : N Nodes
- **Focus on Scalability** – Client code

- Nodes added by simple replication, client should adapt to multiple targets with little effort

- **Applications**
  - Distributed control & monitoring (small scale)

- **Potential Concerns**
  - Synchronization of node data
  - Growing # of unique network streams
Shared Variable Static Nodes vs. Programmatic APIs

- (Node) Scalable systems ONLY use static nodes local to the library location
  - Easy code readability, eliminates need for more complex API

- (Client) Programmatic API allows re-use of same code to read/write variables on multiple nodes
  - Provide unique IP addresses to different instances of the same code
Shared Variable Static Nodes vs. Programmatic APIs
Human Error on adding nodes
Human Error on adding nodes
Timestamp Synchronization (IEEE-1588/Precision Time Protocol)
Time Sensitive Networking (IEEE 802.1AS)

IEEE 802.1AS-compliant Switch
Server-less Architecture

- **Communication** – N Clients : N Nodes
- **Focus on Scalability** – Client & Node code

  - Client should adapt add’tl Nodes w/ little effort
  - Usually implemented with Shared Variables

- **Applications**
  - Multiple clients for simple distributed system
  - Any client can talk to any node

- **Potential Concerns**
  - Load on nodes due to multiple client access
Shared Variables - The Sweet Spot

- Monitoring – Current value (tag) data
  - If you can’t afford to lose some data, don’t use Shared Variables

- Control – Non critical / non time-dependent settings
  - If a malfunction or safety issue can be caused, don’t use Shared Variables

- Slow Data Rates (< 10 Hz)
  - If you need to go faster, don’t use Shared Variables
Scalability Red Flags with Shared Variables

Shared Variable Updates per Second = \# of Shared Variables \times Update Rate (Hz)

The relationship between CPU load and shared variable updates per second are shown for each target in the following figures.
Shared Variables – Scalability Tips

- Update variables only when needed
  - Update variables at different rates in different loops
  - Mass updates of variables puts unnecessary load on system

- Subscribe only to the variables you need
  - If you only need to read a few variables, don’t subscribe to all of them

- Use small, well organized libraries
  - It’s easier to deploy, update, subscribe and read variables like this

- ALWAYS benchmark your worst case scenario
  - There are many factors that affect system performance with variables
Single Server Architecture

- **Communication** – N Clients : N Nodes
- **Focus on Scalability** – Server code

- Nodes can be added by replicating & configuring server
- Any client can connect without the server having prior knowledge of it

- **Applications**
  - Multiple client nodes for complex distributed system
  - Any client can talk to any node

- **Potential Concerns**
  - Load on server due to multiple client access
Offline Waveform Processing

Advantages

- Eliminates unnecessary constant streaming of data to clients
- Utilizing non-volatile storage on the server frees up disk space on the nodes
- Server can use NI solutions like Data Finder Server Edition or 3rd party solutions to aggregate data from many nodes

Concerns

- Network bandwidth sending large volumes of data from many nodes to server
Message Broker

Advantages

▪ Allows any client losslessly send messages or receive a response from any node while minimizing # of communication channels
▪ Server is responsible for keeping track of where nodes are on the network
▪ See Advanced Message Queuing Protocol (AMQP) and applications like RabbitMQ (or LabbitMQ for LabVIEW)

Concerns

▪ Scalability burden entirely on server
▪ Creating functional and scalable server code can be challenging
▪ Multiple clients simultaneously accessing same node
STM toolkit

- Reduce complexity for designing a network communication system for distributed embedded client.
Example: NI SystemLink

Manage distributed systems with web application software for mass software deployment, device management, and data communications.

- Nodes: LabVIEW API for data communication (Tags, Messages, Files)
- Clients: Web application, LabVIEW applications, NXG WebVIs, Data Dashboard
- Server: Interact with LabVIEW API, Data storage
SystemLink Architecture

High performance, secure data services for tags, messages, and files.

SystemLink Server

Tag Service

Message Broker

File Service

Live Data Dashboards

LabVIEW NXG WebVIs

Web Browser

Intuitive LabVIEW and HTTP APIs

Data Service APIs

LabVIEW

LabVIEW NXG

C, PYTHON, OTHER

• NI Controllers
• PC-Controlled Devices
• 3rd Party PC Nodes
Multi-Server Architecture
Summary

- Select the right communication mechanism
  - You probably will need more than one!

- Minimize coupling between components
  - Maintain as few connections as possible, offload routing to servers

- Configuration should be easy
  - Remove the need for configuration or automate it with config files

- Minimize unnecessary network traffic
  - Send deltas when possible, only to whoever NEEDS it

- Use timestamp synchronization when appropriate