n-Tier Architectures for the Creation of User-Friendly Systems

Abstract

All too often, data acquisition and control systems are designed to be easy to use by only one person -- the designer. By the proper application of n-tier software architecture and a few simple rules, a LabVIEW application can be constructed that is not only easy to use, but obvious to use. N-Tier architecture, commonly used for distributed applications, lends itself to data acquisition and control systems because it allows for the separation of interface, data collection, data storage and control into completely separate functional blocks. Though this design requires more work up front than a traditional LabVIEW application, it makes customization of the user interface nearly trivial. With the distributed-intelligence model in LabVIEW 8, this programming paradigm has become simpler to implement.

Introduction – Bad UI's

Though no one wants to admit it, at some time in the past we have all written a code with a bad user interface. We delude ourselves into thinking that it doesn't matter, but we all know that it does. We ourselves don't want to use a system with a poorly designed interface. In cases of extreme awfulness, a system can go from unusable to downright dangerous. As an example, if your system is an airplane and one engine suddenly bursts into flames, you don't want to be told by a small green light somewhere off to the side. "Oh, by the way, I do believe your engine is on fire."

Now there are a few reasons that we create bad user interfaces. Some of these are valid and some are not. Before going into how to fix it, I'd first like to explore how it got to be bad in the first place.

I don't know how to make a good user interface.

This is reason zero. If you are relatively new to LabVIEW programming, you probably have many more things to worry about than crafting a good interface. Following this presentation, this will (hopefully) be less of an issue.

There was just no time (or money).

This reason is somewhat excusable. A well-crafted UI takes time to create. And when you are on month fifteen of a six-month project, some things have to give. However, if you begin the design with the UI in mind, even this doesn't have to be the case.

I'm the only one using it.

This is also somewhat excusable. An always-valid instance of this excuse is the case of the sub-VI. If no one (but you) will ever see the user interface, then it only has to be coherent – not pretty. Another instance is if you are the sole end-user of the system. This is not a good excuse because code often outlives the original user. Please follow the golden rule and leave your heirs a pretty UI. Remember: Code un-for others, as you

would have them code un-for you. By the way, this goes for documentation also. 'Nuff said.

There is no processing overhead available.

This excuse is actually a fairly common one. Face it: 3D controls and rendered graphics require a fair amount of computer to create and display effectively. Often we (in the data acquisition and control realm) are saddled with processors that were inherited from generations past. I personally see my fair share of 486 and Pentium I machines running Windows 95. As far as good graphics go, they just can't hack it.

Now for the caveat: a good UI doesn't need good graphics. Let me repeat that for good measure. A good UI doesn't need good graphics.

Sure, we all would like our systems to look like something out of a science fiction movie, but that level of graphics can hurt more than it helps. As a case in point: one of the best systems I've used is my menu-driven iPod. If it had animated icons and complex screen backgrounds, it would quickly go from elegant to annoying. Graphics are just not important when it comes to UI design. Nice? Yes, when done properly. But you are more often judged on usability and style rather than graphical brilliance.

I have no sense of style.

I'm afraid I can't help you on this one. You have probably all seen VI's with lime green backgrounds, magenta text and orange graphs. I'm sure the designers thought they were gorgeous. My only comment to this: Just say no!

Seriously, if you need help in designing a good user interface, then here's a couple of references:

The Macintosh User Interface Guidelines. And if you can, find a copy of the circa-early-90's edition. This should be on the library of every system designer regardless of what platform they are using.

Edward Tufte, *The Visual Display of Quantitative Information*. This is a true masterpiece and is a graduate-level course in human-machine interaction in a single book.

Introduction – Good UI's

Before this becomes a user interface bashing session, let me explore some of the good interfaces that exist in the world. Most of the examples (at least the ones that are relevant to us) fall into one of these categories:

1. Consumer electronics
2. Medical equipment
3. Web sites
4. Video games
5. Science fiction

There's a simple reason as to why good user interfaces are more common in these environments: evolution. There is intense market pressure to create user interfaces that are clean, powerful and elegant for each of these categories.

First, consider consumer electronics. The iPod is the number one music player in the world because it manages media better than any other on the market. In the consumer electronics game, the market forces the cream to rise to the top. In holding to the philosophy of keeping it simple, Apple created a device that is not only simple, but obvious to use. Though your grandmother may never use one of your LabVIEW VI's, *I* might and I don't want to read a manual!

Medical devices are very much like consumer electronics, but with one important twist: when used improperly, people die. This is perhaps the most powerful motivator for good design. No, not death, but fear of lawsuits. The next time you are in a hospital, look at an EKG machine. Its displays are laid out such that even an untrained user can immediately glance at it and get a good idea as to the condition of the patient.

Next, look at successful web sites. The Amazons and Googles of the world got to where they are because anyone could immediately figure them out. This sort of evolution is like that for consumer electronics, but with a much shorter time scale. Because the web designers could tweak things quickly, well-designed sites became popular and popular sites rapidly converge on an optimal design.

Finally, look at the state of video games and science fiction movies. This may seem odd, but consider that the high performance gaming systems of today will become the mainstream computers of tomorrow, and downright low-end in five years. These systems will be the ones that we inherit for our future designs. Video game (and science fiction show) evolution is similar to both web site and medical device evolution. The systems can be changed rapidly, the industry is rewarded for good design, and someone's life (or virtual life at least) is contingent on their proper use. Note that I'm not advising you to make a Quake-like data acquisition system (though I'd love to see it if you do), just borrow relevant principles where they make since.

As this last point is a bit vague, let me give you a real-world example. A classic first-person shooter is the Marathon series of games from Bungie. Some of you may remember it. In the game, you are a space marine defending your ship against aliens. As part of your user interface, you have a "radar display" that lets you see where the good and bad guys are. Early on in the development of the game, this radar screen presented different shapes and colors depending on the threat distance, threat level, type and number. In play testing the game, this was too much information for the player to quickly process. In the final version, the radar screen display was modified to the much simpler: red = bad guy, green = good guy.

n-Tier Architecture

Now that we have some understanding as to what we need to do, I'll go on to explain how we can do it. All of the aforementioned "good UI" systems have one thing in

common: their user interface is abstracted and separated from the processing components. This is an example of what we call “n-tier architecture”. This is best illustrated by example.

The classic example of this abstraction is that of a web site. A web site is often designed as a three-tier system. In the first tier is the source of the data. This could be a database, a file server or another program performing real-time processing. The data is then sent to the second tier: the web server. This piece acts as the intermediary: it takes requests from the user, fishes out the data that they want from the source, packages it up and ships it out to the third tier – the user’s browser. Each of these tiers is a complete, functional and totally separate system from the others. If the web master wants to change the content of the site, he or she only has to change the data at the first tier. The web server and browser don’t even have to know about the changes. Likewise, the user upgrading their browser doesn’t affect the other tiers in the slightest.

Let’s look at another example. In the aforementioned video game, control, display and model are completely separate functional blocks. The control sends instructions to the model. The model then makes decisions based on those instructions and then passes the results on to the display. The display is responsible for translating the virtual space represented by the model into what the user sees. Again, these systems are completely independent of each other. A change in one has no effect on any of the others. This is why game companies can use each others rendering engines and why you can play many games with keyboard, mouse or game controller.

By definition, an n-tier system is “… an architectural design pattern consisting of multiple levels of independent processes, each of which passes data to its nearest neighbors.” This definition captures the essence of my point: that these multiple independent processes (or threads) can simplify your programming, allowing you the time and resources to create better interfaces. Now that you're getting the idea of what n-tier architecture is, we'll look at what it means for our data acquisition and control systems.

Standard data acquisition system design

Most data acquisition systems are designed as monolithic entities. Their acquisition, control and display components are bundled together into a single main virtual instrument. Sometimes these functions are encapsulated into separate sub VI's, sometimes not.

There is a single compelling reason for this type of monolithic architecture: it presents an obvious solution to simple data acquisition tasks. Most of the applications that we create are variants of the familiar oscilloscope. You acquire a block of data, display it, process it and save it. That's it. If you program LabVIEW for a long time, you may build up a library of VI's to streamline common tasks, but you still tend to reinvent the wheel a lot, creating VI's that are variations on the same theme. The monolithic design is common (even shipping as examples with the LabVIEW environment) and can easily be adapted to meet a variety of different needs.

Is this the best way to code in LabVIEW? For very simple applications, it certainly is. If however your application is more complex than a simple n-channel oscilloscope, you will quickly exhaust the effectiveness of this simple programming model.

In point of fact, it doesn't take much in the way of complexity before the monolithic model begins to break down. Once you get to the level of a few hundred sub-VI's, you better have either a very good plan or very good architecture. Somewhere around VI number three-hundred, adding functions to an existing application becomes an exercise in refactoring as each new link threatens to break an old one. You would think that there would be a clear path from the monolithic model to the multi-tiered one. You're right.

Breaking the monolithic model

Let us start to abstract the basic monolithic design a bit. I made brief mention of a couple of sub-systems that can be re-factored into reusable components. You may have already taken this step in your own refactoring efforts. Let's take our oscilloscope example from before and look at the individual components one by one.

1) Acquire
2) Display
3) Process
4) Save

Data acquisition is generally a task that requires a high-degree of customization based on the nature of what you are trying to measure. Often we're faced with a mixed signal environment in which our job is to read in both analog and digital signals and maybe even a counter or two. This will probably be the last piece that we standardize. There's just no way around that one.

Data display, in contrast, does lend itself to some standardization. After all, how many different ways to display data are there? For our hypothetical oscilloscope, a graph is the most practical means of displaying acquired data. We have many different types of graph to choose from, of course, but realistically, how often do you use more than a strip chart, scope chart or X-Y plot?

Data processing, like data acquisition is highly dependent on the nature of the measurement. In fact, we can probably lump this chunk with the acquisition chunk and do them both at the same time.

Saving data is the first part to standardize. This is the most important piece for three reasons: First, standardization of data storage means that you can create a library of inter-operable tools to manipulate your data. Second, using standard data storage components free up a considerable amount of your time in future projects. And third, there are probably only a few ways that you will ever want to store data. In general, I can think of only three formats you will ever need:

1) The ubiquitous something-delimited-text format
2) The blazing-fast binary data format

3) The file format for some third-party software vendor that you hate but others at your company seem to love (as an example, for me that would be the Matlab binary data file format (.mat))

When you create a standard library of VI's that are largely interchangeable for each of these file formats, you life gets much easier. Obviously, applications will differ slightly, but you can still share the basic functions of open, write, read and close.

Now that we see how some of our modularization can work, we'll make a conceptual leap into a true multi-tiered application.

A closer look at multi-tiered applications

The one thing that all multi-tiered applications have in common is a communications infrastructure. This can take on many forms. In the case of a web-application, it is the TCP/IP protocol that forms the backbone. In a game, it is the message passing interface of the programming language itself, object calls and methods in C++ for example. For something like a piece of consumer electronics, the communications backbone can be non-obvious. The I2C standard amongst board-level components in a cell phone is one example. MIL-1553 in an aircraft is another. Obviously, we can have many different levels of communication occurring simultaneously.

The communications backbone is really the heart of an n-tiered system. Indeed, it is what makes it n-tiered in the first place. It is responsible for all of the "action" that takes place in the application and manages the data flow between the various components. To better illustrate, let us continue with our oscilloscope example.

We have created three modules. The first is responsible for data acquisition and analysis. The second handles data display and user interfacing. The third is responsible for data storage and retrieval. How do they talk to each other? In most LabVIEW applications, this is very much tied to the application itself. We would have a few lines for control signals, a few lines for array-based or waveform-based data and perhaps a cluster for constants or properties that we want to keep track of: error messages, file locations and the like. Though fast and efficient, this means that we have a lot of data lines to try to keep track of. And heaven forbid that we want to make a fundamental change to the user interface. Such an undertaking would require major reworking of every component in our system.

Why don't we rip all of that wiring out and replace it with something that we can standardize? Because that is precisely the wrong way to look at it! And no, I'm not suddenly reversing my position. Standardization of VI-to-VI communication is a wonderful thing and can be a very powerful technique, but if it were so great, it would be a part of the LabVIEW environment already. There is a reason that there are so many different types of inter-VI and inter-process communication available.

Instead of looking at creating a Swiss army knife of inter-VI communications, it is more efficient to tailor your communication backbone to something that you can use without bogging down your application with a lot of overhead processing. Sending, receiving

and processing messages can be costly in terms of CPU time and isn't something that you can universally apply.

Of course, if your applications are all highly distributed then standardizing on one type of communication line becomes appropriate. This is commonly done in industrial SCADA systems. These systems can have thousands of sensors and control points, but their updating is limited to the speed of the user's brain. In such an environment, if any one gage gets updated every tenth of a second, it is more than fast enough because each single node in the system handles its own high-speed acquisition, processing and storage. All the user needs in this instance is a (relatively) low-speed overview of what the larger system is doing.

A related exception to the Swiss army knife rule is when all of your acquisition processes are very high or very low speed. Always remember that "real-time" is in the eye of the beholder.

If you are trying to capture events from a particle accelerator or fusion reactor, there is nothing on this planet that is going to be fast enough for you. At this highest level of data acquisition kung fu, you are building your own giga- or even tera-sample acquisition components. If this is your application, your hardware had better handle all high-speed tasks (and will probably be cooled with liquid nitrogen). The LabVIEW component then looks just like it does for the previously mentioned SCADA system and standard communications again looks feasible.

The other extreme is if your definition of a "real-time" application is measuring plant growth or weather conditions or something similarly slow. In these examples even one point per second is probably too fast. In this case, you may as well use a single backbone, as you will have more than enough time for any communication-processing overhead.

Now for the meat of this presentation: designing backbones for inter-VI communication.

Multiple communications systems: a practical solution

There are a few different communications schemes from which we can choose. Our decision will be based on a few factors: processing speed, message complexity, system distribution, etc. First, a high level overview of each type of system:

Option 1: Queues

Queues have several distinct advantages over other means of passing messages between processes and objects. First, they are simple to implement. LabVIEW has a number of built-in queuing functions that are easy to understand and easy to use. Second, their behavior is deterministic (important if you are aiming for real-time) and they translate well to a variety of systems, including embedded platforms. Finally, they are capable of handling any arbitrary data type.

Before the LabVIEW gods jump down my throat, I'd like to issue one caveat. Queues are not necessarily deterministic in the real-time sense. In point of fact, most of LabVIEW is not necessarily deterministic either. However, even though the built-in queuing tools cannot be used in LabVIEW Real Time, Embedded or FPGA, a programmer can construct their own queue handlers with all of the properties of LabVIEW's built in queuing functions.

The greatest disadvantage of using a queuing system for communications is that queues cannot span multiple systems on a network. In order to be used as part of a networked environment, a queuing "gateway" would need to be built in order to synch the contents of platform-specific queues across a network connection. As we shall see, this is not so terrible a limitation.

Option 2: TCP/IP message passing

TCP/IP message passing is, in many respects, the opposite of queues. Whereas queues are easy to use, deterministic and can use rich data types, TCP/IP is prone to obscure bugs, can have high latency and everything must be converted to and from string data. The advantage that TCP/IP has over queues is that, by its very nature, it supports a high degree of distribution across a network.

TCP/IP has the added advantage of being nearly ubiquitous. If you need to fit some custom hardware into your distribution scheme, odds are high that it supports TCP/IP out of the box. With a bit of configuration and programming, a LabVIEW based TCP/IP message passing scheme can be made to fit with all sorts of third-party systems.

Option 3: Global variables

By now we've probably all heard the mantra: use global variables sparingly. For the most part, this is true. Global variables are simple to use, but their simplicity belies an inherent inefficiency. Other communications protocols are simply faster. But let's take a closer look anyway.

Like queues, global variables are simple to implement. You can use them exactly as you would a queue, but without the hassle of remembering how things stack and the ability to access them freely at any point in your code. They support many different types of systems (though are not as universal as are queues) and can handle data types of arbitrary complexity.

Now, with the advent of LabVIEW 8's "distributed intelligence" model, global variables gain the advantages of TCP/IP message passing. Yes, they still have some processing overhead associated with them, but at the same time, they gain several advantages that TCP/IP systems do not have.

Global shared variables or (for sake of clarity) "distributed variables" are as easy to use as traditional global variables and, like them, can be read or written at the point where they are needed in the code. In addition, there is little performance hit from using them (though don't go to crazy as there is some hit), and they can support arbitrary data types

(because they are still regular global variables). Finally, they use a publish-subscribe network model (also sometimes referred to as "broadcast").

To go off on a slight tangent, traditional TCP/IP uses a talker-listener model. When setting up a TCP/IP connection, a particular talker pipes data to a particular listener. There is no broadcast option and multiple listeners would have to pass data between themselves (as with things like Bittorrent). In contrast, the publish-subscribe model supports broadcasting as one node on the network can write to the distributed variable and many systems can read from it. This has more processing overhead associated with it, but LabVIEW takes care of that for you.

Option 4: Something else

Yes, there are many other sorts of communication systems out there. How often are they used? Not very often at all. Just to address a few of them:

UDP is similar to TCP/IP except that it also supports broadcasting. As such, it gains the power of distributed variables, but at the same time gains the complexity of TCP/IP. In the process it looses one of TCP/IP's big selling points: it isn't universal.

How about DataSocket? Yes, it is easy to use and can support complex data types. In practice however, DataSocket has a good bit of overhead associated with it. I have seen systems that make liberal use of DataSocket bring a network to its knees. Since you have no direct control over DataSocket's network use, it ends up being more efficient to use TCP/IP, distributed variables or even UDP.

ActiveX and dot-Net? Let's not go there. I realize that they promise a lot, but I have never seen a system successfully use either for inter-process or inter-system communications. And I for one would rather not be tied to the whims of Microsoft. Has anyone ever heard of AppleTalk? Is anyone currently designing with it? 'Nuff said.

Now that we are more aware of our options, let's continue with our system design. For our purposes, I recommend a multiple backbone system consisting of a queue for local (same-machine) communications and a TCP/IP message passing system for distributed functions.

Why not use distributed variables? There are two reasons for this. First, how many people here are using LabVIEW 8 yet? And how many of your clients are? If you use two different but connected communications backbones, you gain the advantages of ubiquity and compatibility both forward and backward, though at the expense of some added complexity. Next year, my advice to you will probably be different.

Designing the message

When designing the communications system, it is important to design it around the message you are sending. The message should be able to support rich data types and yet be flexible enough to be translated into strings for TCP/IP transmission. As an example, the message format that I use is as follows (in pseudo-code)

```
struct message {
      string      msgType
      string      msgRecipient
      string      msgSender
      string      message
      string      parameter[]
};
```

This message structure gives me a number of useful fields. The message type is a flag that I use to tell the target what to expect. I can use this to parse this into things like requests for data, control messages, etc. The recipient and sender fields allow me to parse by intended target and sender. With these, I can filter messages by sending machine or process, or look for things intended for a particular process or machine only. In practice, these first three fields are often nulled-out or set to some harmless value like "message" or "localhost".

The message field is the real meat of the message. This is the instruction that is parsed by the recipient. It describes the action to be taken. The parameter array is a list of modifiers used by the message. Note that these are all strings. Though limiting ourselves to a string data type is by no means required, it retains a high degree of versatility as strings can be cast to and from other types.

To further illustrate the use of this message structure, let me give a concrete example. In a highly distributed system, we can expect that each machine on the network will have its own local data. To request a piece of data, you can send the following message (from VI 2 on the machine at 128.232.0.15):

```
query:128.232.0.1_vi1:128.232.0.15_vi2:data_request:analog:temp_0
```

In response, the receiving machine (VI 1 at 128.232.0.1) could send back:

```
data:128.232.0.15_vi2:128.232.0.1_vi1:data_push:analog:temp_0:792.33:F
```

It should be obvious how these messages are translated from the LabVIEW cluster to the TCP/IP string and back. I often use colons to delimit, but anything not commonly used for your data will work also. For processes on the same machine, the message can look like:

```
query:localhost_vi1:localhost_vi2:data_request:analog:temp_0
data:localhost_vi2:localhost_vi1:data_push:analog:temp_0:792.33:F
```

or:

```
query:process_1:process_2:data_request:analog:temp_0
data:process_2:process_1:data_push:analog:temp_0:792.33:F
```

You get the idea. Obviously, the precise details of the message – the keywords and data identifiers – are left up to the end user. In addition, you may need different or additional

fields in you message structure. For instance, a flag for "single target" versus "broadcast" may not be a bad idea.

Sending the message – local queues and gateways

The easy part of messaging is the use of queues to pass it between VI's and process threads on the same machine. Simply set up a named queue to take the message data type. Other VI's and processes on that machine can access the queue to share data and control signals between them.

If messages are few and far between, the queue can even be replaced by a single global variable that contains the message structure. Every process and VI on the local machine can then use this single global variable as a sort of "dead letter drop" to pass control signals and data.

Complexity is added when you want to share the contents of the queue between different machines on the network. It is possible to design a TCP/IP based system to continuously synchronize local queues, but in practice this can be overkill. Such a shared queuing system can increase network traffic to an unmanageable amount and is rarely needed anyway.

A better way to do it is to cast and catch messages sparingly, inserting them into the local queue as they come in. This allows us to decrease the amount of network traffic dramatically by cutting out local-only messages that shouldn't be sent to other machines.

Is this all there is to it? Not really. In addition to the queue and the gateway, you will probably need a number of global (but not distributed) variables. These global variables will act as buffers to handle acquired data and perhaps some state information. They are for convenience and to allow us to cache acquired data so that it can be shared with other components on the local system. The reasons will become apparent shortly.

The basic system

The creation of our distributed system will begin with the creation of the primary component. This component will have one function and one function only: to pass messages and to launch the other components. My sample starts by initializing itself and by setting up both the queue and gateway. Other components are launched as needed in response to user instructions. In this simple example, I have three other components: the DAQ demon, the file demon and the display.

The DAQ demon is (as it's name implies) a processor thread that sits in the background, gathers data from the sensors and pipes it to a series of global variables. The file demon grabs the data from the global variable and writes it to a file. The display component grabs the same data and displays it in a user-friendly fashion. And here we see the reason for the aforementioned global variables. If all acquired data were passed via the queue, we would severely limit the update rate of the DAQ demon. By writing the data to a global variable, we are freeing the queue for the more important work of handling control messages.

So why then do we need messages? They govern the behavior of the different processor threads.

Going back to our example, the user starts the system by opening the main VI. The system goes through its initialization process, starts the VI's for the DAQ and file demons, then sits idly by and waits for instructions from the user. Let us assume that in our simple example, the user has the following commands at their disposal (perhaps triggered by menu items or buttons on the main panel):

Configure the data acquisition component
Start and stop data acquisition
Start and stop file save
Pull up a graph of the data
Quit

The first instruction pulls up a dialog asking for a configuration file. If the user selects one, the main component then sends the following command to the DAQ demon:

```
control:local_daq:local_main:configure:<selected file path>
```

The DAQ demon goes through its configuration process, setting up data acquisition as specified by the selected file. When ready, it sends back the following:

```
control:local_main:local_daq:configure:ok
```

Now assume at some point in time in the future, the requests data acquisition start or stop or that the acquired data be saved to a particular file. The messages for these operations could look like this:

```
control:local_main:local_daq:start
control:local_main:local_daq:stop
control:local_main:local_file:saveto:<data file path>
control:local_main:local_file:closefile
```

The user interface component is a separate VI that is opened in response to user request and grabs the data from the buffer (the global variable) and displays it. It may or may not have its own set of control messages that it sends or receives.

The final message, the quit command, might send a message like this:

```
control:local_main:local_all:shutdown
```

The main component then quits following the shutdown of the other processes and closure of their VI's.

Now we'll add a bit of complexity to our system. Suppose the DAQ component makes use of a state variable on another system – a piece of calibration data for one of its

sensors, perhaps. The DAQ component can make the request for that data through the main process as follows:

```
remote_query:128.232.0.15_main:local_DAQ:cal:analog:rtd-0:scale
```

The main thread picks this up and passes it to the remote machine after inserting the local host's address:

```
query:128.232.0.15_main:128.232.0.1_DAQ:cal:analog:rtd-0:scale
```

The information then comes back from the remote machine:

```
data:128.232.0.1_DAQ:121.232.0.15_main:cal:analog:rtd-0:scale:0.523
```

And then is passed to the DAQ demon so that it can properly scale its sensor based on the stored calibration data.

Something Missing

If you've come to this point and think that there is something missing, you are absolutely correct. We've discussed the transfer of high-speed data, low-speed data and control messages. What we've neglected thus far is the transfer of configuration information.

The normal method of storing and passing configuration data is by the use of clusters. One (or more) cluster holds all configuration variables and is passed to each and every sub-VI that might need them. The cluster is generally saved as a control template such that changes to the cluster are propagated throughout the system.

This has two drawbacks. First, as the cluster grows in complexity it becomes increasingly difficult to see everything in it. Yes, it is possible to use tab controls to increase the effective space. You can make font sizes very small and compress everything to near-unreadable. This doesn't solve the basic problem of adding just one more control.

Second, it may not be possible or practical to save the cluster as a template control. Development of the sub-VI's could be in several different hands or done at different times. This would lead to the dilemma of breaking the VI in order to add another variable to its configuration data.

The solution is to maintain all variables in a structured list. This is generally a two-dimensional array of data such that one column represents the data tag and another the data itself (with perhaps a third for the data type and/or a fourth for the units). In practice this would look something like this:

DAQ_AI_rate	12.5000	dbl	kHz
DAQ_AI_gain	20	dbl	dB
DAQ_AI_offset	1.2345	dbl	volts

This simplifies the process of passing configuration data as only one structure ever needs to be passed to any VI. The individual sub-VI's then can read and modify the data that interests them. By using a standard library of functions, this process can be as simple as accessing variables in a cluster by name.

Since this is just a two-dimensional array of strings, it is simple to read or write to a file, or to pass between remote VI's via TCP/IP, queues or distributed variables.

Addition of advanced features

This basic system is just that: basic. There are very few tests where we can get away with writing a simple oscilloscope-type application. After all, if that were the case, you would be perfectly happy with the examples that ship with LabVIEW.

Suppose some additional data processing is needed. In this case, the simple DAQ demon can be removed. In its place, you can add a VI that handles the data processing you need in addition to the acquisition functions. So long as it handles the same messages (with perhaps a few additions) and the same configuration file format, it is a drop-in replacement for the basic DAQ demon.

Suppose new file formats are to be supported. The DAQ system does not need to know anything about them as it just pipes data to the buffer. A new file demon can be written that handles additional formats yet still responds to the same messages as the old demon.

Perhaps your system needs to output data as well as acquire it. You can create a new output demon that sets analog and digital lines in response to either user control or to the values of data on the buffer. The user interface components or DAQ demons can send messages to the output demon as well as transferring data to and from the buffer. Again, no other piece of the system even needs to know of its existence.

Multiple demons of the same type can be running at the same time. If you have a set of sensors that you read for every test, then create a common DAQ demon for them. If another sensor or two needs to be added, then they can have their own demon that uses the same, shared data buffer. In this way, tests that are variants of a base test can be conducted easily.

And of course remote messages offer a world of possibilities. Picture another process that acts as a database for calibration or test data. DAQ demons located throughout your plant can query them for calibration settings or send them data to use in enterprise-wide reporting. Again, these features can be added without disturbing the operation of other components on the network.

Even the main component can be updated with no change to the other pieces. If you want to publish you data to the network, the main component can broadcast data channels to a central server. Other messages can be added to handle things like remote login, web publishing of data or automatic report creation.

And the point of all this is …

Herein lies the power of this system. The main process, the DAQ demon and the file demon can be reused for nearly all of your data acquisition tasks, with only minor (if any) tweaking needed. Yes, they require a significant investment of time, but their reuse can save you a great deal of time in the long run.

The user interface components only require a small amount of programming as they only pick data from the buffer and display it in a meaningful fashion. In fact, you can build a standard library of user interface components and reuse them as practical. Now that lower level functions are standardized, you will have time to create those powerful and good-looking user interfaces.

The power of n-tier architecture will become apparent the first time someone tells you something like, "…you know, I'd really like to see that particular sensor on a separate graph with maybe a running average". Rather than re-working every level of a monolithic VI, strip out the old user interface component, make a modified one in a matter of minutes and drop it back in to the existing application.

And don't forget to pretend that it will take all morning when asked. We all want to seem like miracle workers, of course.