ELEMENTS OF CONTROL SYSTEM LONGEVITY

Stephen A. Lewis, SLAC, Menlo Park, CA 94025, U.S.A.

Abstract

We build controls for long-lived facilities. What are the essential architectural elements that are likely to give any particular approach a long tenure? Many aspects can easily be identified by their negative value, such as: dependence on particular languages, operating systems, or hardware. I will argue here that the fundamental positive aspect that gives the greatest endurance is "decoupling, decoupling, decoupling." The principle of decoupling applies in many contexts. I will attempt to show that among the key contributors to achieving this desired state are very stable, very narrow 'intellectual' bottlenecks at appropriate levels; decentralization; and the use of asynchronous communication.

LONG TENURE

We design, build, and maintain control systems that are often specified to last 30 years. Many of them have lasted that long and far longer. But in 'computer years,' 30 years span many technical generations. Thus, we should consider in the design phase how to mitigate the cost, effort, and even disruption that follow the inevitable need to upgrade. If you are tasked with a first version of your control system to test the first stage of a 5-year construction project, you could be doing your first upgrade before facility is commissioned! In the end, it will not be the qualities of the components that fails you: it will be the dependencies among them.

I am guided in my following remarks by two aphorisms that my own personal history has given much credence to:

> "*In theory, there is no difference between theory and practice. But in practice, there is.*"[1]

> "*Success comes from experience; but experience comes from failure.*"[2]

PITFALLS

Many of these topics, although technical, carry a heavy emotional burden. One such topic that I won't specifically address is proprietary *vs.* open-source choices—the issues are too complex and open-ended for this paper.

Language

The choice of programming language is often based on very abstract or conceptual concerns. In our industry, however, the language *du jour* is quickly replaced by another. And so the short-term *developer's* joy of having that special language is soon replaced by the long-term need to find trained *maintenance* programmers, compilers for current hardware and OS platforms, and the graphics, communications, and mathematical libraries for all the functionality your site depends on. You do have to select one or a few languages, so avoid depending on features that do not have equivalents in other 'main-stream' languages. But language choice is much less important if the total code size can be significantly reduced (see *Requirements* below).

The language topic, nowadays, always brings up the question of choosing an Interactive Development Environment (IDE). Certainly, if you are going to aggressively develop and maintain mega-code, you are going to need all the help you can get. On the other hand, almost no aspect of modern software practice seems more volatile than the IDE area. The particular danger to avoid is to find out that no compatible combination of that IDE and your new language, operating system, and host hardware exists.

Operating System

Discussions about different operating system have often been described as 'religious wars.' Like programming languages, operating systems come and go. Finding experts to keep an ancient OS going is hard, and an ancient OS typically requires old hardware—and a store-room of spare parts.

In addition to selecting features that are generally supplied in a typical OS (especially in the real-time area), an excellent practice is to provide a *thin* 'glue' layer between local code and the OS services.

Transport

The transport or 'middle-ware' elements of a control system form its back-bone. Hence its features, and efficiency (or lack thereof) can make or break the scalability of the entire installation. These days, most are built upon the socket interface to TCP/IP and so ultimate portability is not the driving issue. Perhaps a greater pitfall is the case where it distorts your architecture (see *Asynchronous*, below).

Concepts such as name discovery, graceful failure and recovery, congestion controls, and so forth are best defined independently and then carefully 'mapped' onto the underlying services. This choice must also be carefully coordinated with the choice of *wire protocol*, discussed further below.

Hardware

Hardware has undergone perhaps the most dramatic changes during the four decades we have been building control systems: from mainframe, to minicomputer, to workstation and crate, to 'smart' device. The network has changed to keep pace: from a stand-alone computer systems, to a simple point-to-point star or ring, to a local area network, to a 'cloud' of switches and routers. The number of layers has varied between one and three.

For crates, the 'bus wars' have never truly abated: CAMAC; Multibus I and II; VME; and now ATCA and Mi-

croTCA, to name a few, will likely be mixed in many systems.

With care, I think you can mix stages with different numbers of layers and so evolve gracefully. The key is not to *assume* too much about how various entities are distributed in layers (and see *Layers* and *Wire Protocol* below).

Institutional Aspects

We build our systems in an institutional matrix. Its culture can intrude on our design choices. My personal and anecdotal observation is that there is a strong tendency to follow the organization chart in the overall topology—hence the popularity of 'manager' or 'supervisor' or 'director' entities, sited just where their human counterparts would be in the block diagram. I prefer a more 'direct-connection' model (see *Decentralization* below).

Operating system and platform choices are often highly constrained by institutional standards and practice (at least by its IT department); my advice is to make your technical case and fight very hard. Your needs will follow the slowly changing facility control needs, whereas the institutional constraints will follow the rapidly changing regulatory, funding, and senior management rules—quite a mismatch.

DECOUPLING

It is the interplay of assumptions, constraints, and design choices that can make an upgrade onerous. The antidote, I assert, is *decoupling*. (A close relative of decoupling is *decentralization*; see that below.) Decoupling comes in many forms…

Layers

Do you know where your layers are? An architecture with separate communicating entities does not automatically have layers; and you can have entities with dual roles (say, publisher and subscriber) but still have layers. I think of it as the 'one layer/one protocol' rule: you are doing it right if the *one* protocol you need to connect two entities is obvious. But there can be too much of good thing: certainly no more than a few protocols should work (and see *Wire Protocol* for why you can have fewer than you might think).

Technical

I refer here to the absence of a 'ripple' effect: a change in one layer should ideally have *no* effect on the layers above and below it—and the more dynamic the process is (the later the binding), the better. With care, you can support multiple versions both *vertically* (old callers can interoperate with a new callees), and *horizontally* (allows mixed versions in a layer).

Social

Social decoupling refers to situations where software developers and maintainers, controls system users (such as operators, technicians, engineers, physicists), and even managers, can go about their duties once they have learned the basic rules. Although the costs of a highly coupled system are seldom quantified, they are evident in the great reluctance to change anything because so many different classes of people who interact with the control system can't or won't accept the perceived costs.

Requirements

Simply put, don't try to turn a 1000-page requirements document into 2 million lines of 'traceable' code. The requirements you will have early enough to meet the schedule will be mostly invalid by the time you finish. Some practitioners estimate that as little as 4% of the requirements are stated upfront [3]; in fact, most are implicit. A more promising approach is to build a collection of reusable building blocks and then track the requirements by changing the parameters and interconnections among them—a combinatorial bonanza. This might eliminate the necessity of mega-code.

KEY 1: 'BOTTLENECKS'

It may seem contradictory to suggest that a bottleneck is beneficial in a control system (and I will make the case against the traditional bottleneck below); but I am referring here to *intellectual* bottlenecks: choosing critical places in the design and implementation where you can use a 'narrow' rather than a 'wide' approach.

Wire Protocol: I

The 'wire' protocol and its associated Applications Programming Interface (API) is what the middle-ware must implement in our current 'publish/subscribe' or 'client/server' paradigm

The 'wide' version appears to provide a boundless future of more and better functionality; and initially it allows many parallel activities as each work-group refines how its communicating entities are used. Yet ultimately, I believe, it becomes more and more burdensome for two main reasons.

First, the intellectual challenge of quickly and correctly choosing all the right 'calls' from the vast number offered actually reduces productivity—especially for new or occasional developers—and leads to fewer and fewer reusable patterns. And when the inevitable follow-on phases occur the burden on every *caller* to adapt to the added or changed syntax and semantics of every *callee* will be almost unmanageable ('version hell'). This is only exacerbated by the parallel work-group activity so useful at first.

Second, a wide protocol/API practically stymies any straight-forward implementation of 'tools,' that is, generic client or subscriber applications. Because the 'narrow' API seldom changes (and may even support older versions concurrently), a tool-style application is mostly decoupled from the versioning process: unlike its 'wide' counterpart, it does not have to track the ever-growing (and ever-changing) 'sea of devices' that any successful facility will be adding and replacing as the physics and technology demand.

Don't underestimate the multiple benefits here: the tool maintainers don't really need much knowledge of the devices; the device experts don't need to tell the tool maintainers what they are up to; and the tool users (operators, technicians…even managers) are experts in using the tools (which are quite stable) and can go about their business with minimal interaction with either group. (This is a nice form of social decoupling.)

Wire Protocol: II

Consider an architecture in which a chain of communication might exist: an entity that drives a sequencing operation might depend on a data manipulating entity which in turn depends on basic sensor/actuator entities. This means there are three layers and two protocols, right? No.

With some care, a single protocol is enough. Essentially, client/subscriber entities can also be server/publisher entities and reuse the protocol. An apparent side-effect that in fact becomes an enormous benefit is that all of the 'observers' that will aggregate around such working chains—synoptic panels, data loggers, alarm notifiers, *etc.*—can be the same single-narrow-protocol tools previously referred to.

But let's carry that logic one step farther: resist the temptation to embellish the client/subscriber tools; keep them as 'thin' as possible. For example, do not add features to allow data calculations within a synoptic client. Instead, use or create a general entity that can parse useful formulas, subscribe to the inputs and publish the output. (Instances of such entities can exist almost anywhere.) Now, not just the specific instance of the synoptic, but every instance of every client entity can subscribe to that new output as just another named item in the 'sea' of published items. This technique can substantially 'flatten' the logical view of the control system, decoupling it from the actual distribution of entities. Such an approach can make even sweeping re-organizations of where functionality is deployed transparent. I recommend using this very powerful approach when implementing Manager, Supervisor or Director entities, despite the initial tendency to give them 'special' protocols.

File Protocol

I think the best approach here is two-fold: first, 'buffer' to a real file any stages in which large amounts of relatively 'slow-moving' data are being moved (such as startup parameters or snap-shots); second, use a clean text (ASCII) representation.

This very strong decoupling in time and format gives you many advantages: you can implement and test the producer or consumer side in any order; you can inspect and/or generate both valid and invalid data using the simple text tools at hand; you can easily keep any or all of the samples in a code repository along with its code; you may be able to 'ride through' a failure of the off-line part of the food chain during critical operations until a repair is complete.

Ideally, one representation (say, XML) can fill all requirements.

KEY 2: DECENTRALIZATION

Decentralization is key for two reasons: it avoids any single point of failure (graceful degradation); it avoids scaling problems as the load or complexity of the overall system grows; it is a natural way to introduce new versions in any layer; and it reduces the likelihood of cascading failures (and see *Asynchronous* below).

Consider using a 'gateway' as an extreme form of decentralization. I mean by this term a dual subscriber-publisher entity that sits astride two controls domains, and selectively 'relays' transactions. Useful attributes here are: allow 'aliasing' to bridge conflicting naming conventions; enforce 'throttling' to prevent busy systems from overloading their neighbors; add additional access rights such as 'read-only.' Finally, should an upgrade at some point entail a shift to a new protocol that is no longer interoperable, a gateway can become a protocol converter (bridge), allowing a phased cut-over.

Location independence is generally considered necessary for decoupling: only 'tags' (names) should used for the rendezvous between communicants. By implementing it with a dynamic 'discovery' protocol or a distributed protocol (like DNS) then decentralization can be preserved.

KEY 3: ASYNCHRONOUS COMMUNICATION

Synchronous (blocking) communications require deep understanding of the larger network of dynamic connections that can occur, and many assumptions about time that cannot be really known and are subject to change as portions of the system are replaced with much faster hardware or the scale of the system grows with time. When there are three or more entities in a wait-for-response chain, they are likely to 'lock-up'. Failure of an intermediate link in the chain requires nearly heroic measures to devise a recovery scheme—sometimes a system restart is the only recourse.

Asynchronous (non-blocking) communications can eliminate these systemic failure modes if correctly implemented. Message queuing also provides an easier path to inserting 'taps' to monitor the protocol traffic.

CONCLUSION

By using decoupling, decentralization, narrow protocols, a flat logical topology, and asynchronous communications, the goals of better scalability, graceful run-time degradation, and uneventful upgrades with changing requirements can all be more easily achieved without heroic measures.

REFERENCES

[1] Variously attributed to Jan L. A. van de Snepscheut and Yogi Berra

[2] Usually attributed to Mark Twain.

[3] R.L. Glass, "Facts and Fallacies of Software Engineering," Addison-Wesley, 2002.