Self-Regulating Software:
Scalable, Predictable, Composable Distributed Systems

Calton Pu

Center for Experimental Research in Computer Systems
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

Main Ideas: The difficulties in building scalable, predictable, and composable distributed systems are well known. We introduce the notion of self-regulating software to address these difficulties. A software module is *self-regulating* if it is able to satisfy a set of explicitly defined Quality of Service (QoS) requirements on its input and output, usually under certain environmental conditions such as sufficient resource availability. The composition of these QoS requirements makes self-regulating software composable. On the systems side, we adopt a single concept: the software feedback, as the implementation technique for supporting self-regulating software. Software feedback is the software equivalent of control system feedback with similarly predictable behavior. We use this uniform mechanism to manage adaptively all resources in the system, including CPU, disk, and network bandwidth. The approach is called progress-based scheduling. We enhance control system analysis to conduct rigorous modeling and analysis of our application of feedback to systems software. The consistent use of feedback makes self-regulating software predictable. On the software and programming side, we use results from two major areas: partial evaluation and program specialization, and domain-specific languages (DSL). We use program specializers to decrease program size and increase program speed, once the code is written. DSLs support smaller programs that are more amenable to optimization. The combination of all the techniques above makes self-regulating software scalable.

Innovative Claims: Building scalable, predictable, and composable distributed software systems is one of the fundamental challenges for software development today. In terms of technology, each one of the technological building blocks we use (software feedback, program specialization, DSL, information flow applications) has been maturing during the last decade and we think it is time to bring these technologies together. In terms of broad impact, Self-Regulating Software will help applications span a wide spectrum of underlying hardware. In networking, the spectrum ranges from gigabit wired networks to kilo-baud wireless networks. In CPU, the spectrum spans from simple PDAs and intelligent appliances to high performance clusters running database and web servers. In terms of maintainability and portability, Self-Regulating Software will help both systems and application software to follow the relentless hardware evolution described by Moore's Law.

Critical Technical Barriers: Although researchers have made significant progress in component technologies such as software feedback, program specialization, and DSL, it has been difficult to bring these techniques together. The most obvious technical barrier is the combination of these techniques, which have been evolving by themselves. This challenge also requires some innovation in each technique, so they can be combined into tools that support the development of Self-Regulating Software. For example, in software feedback, we need to develop better formal analysis of software feedback that are dynamically changing their models and components. Another example is in program specialization, since we need to provide specialization at several program abstraction layers, represented potentially by separate DSLs.

A. Introduction and Motivation

Building software that is scalable, composable, and predictable in any environment is a very difficult task. For a distributed systems incorporating a wide range of system parameters (e.g., fast and slow machines as well as fast and slow networks), this goal seems nearly impossible. We contend that aiming for the apparently impossible is the only hope for developing large-scale Internet applications of the future, other than employing an army of programmers. There are some indications for the difficulty of the goal that we briefly outline here.

Scalability Challenge. Distributed computations and algorithms in general seem to have scalability problems. A good example is the agreement protocols, considered a basic building block of many distributed computations. A very simple example of agreement protocols is the two-phase commit implemented in distributed database systems. Even though the two-phase commit passes around one bit of information (commit or abort of the transaction) and it uses a small number of messages, real world transaction systems rarely turns it on, since it would add considerable latency to transaction execution time.

Predictability Challenge. Given the variety of systems and the wide range of their capabilities, providing Quality of Service (QoS) across a network has been difficult to achieve. An example of promising attempt is the RSVP protocol that supports end-to-end QoS for network parameters such as guaranteed bandwidth. However, even though an approved Internet standard, RSVP has yet to be widely implemented or supported. The current state of art for QoS support is the construction of an entire distributed system under one company's control. By over-provisioning (buying more machines and networks than the predicted load), such a system provides QoS except for peak overload situations.

Composability Challenge. In distributed systems, a property that holds in each location does not necessarily hold globally. For example, consider a distributed transaction processing system. The fact that each component database preserves serializability does not guarantee the serializability of distributed transactions spanning multiple databases. Similarly, we may have individually well behaved control systems that become unstable when put together. Still another example is achieving end-to-end QoS requirements. Suppose that a packet takes 40ms from Atlanta to Chicago, and 80 ms from Chicago to Seattle. If the latency requirement is less than 100ms between any two points, both segments (Atlanta to Chicago and Chicago to Seattle) would satisfy the requirement, but the composition (Atlanta to Seattle) would take too long.

Information Flow Applications. In this proposal, we make the apparently impossible goal feasible by focusing our attention on large distributed applications that are *information flow-driven*. By information flow-driven we refer to applications that primarily process and deliver information, for example, digital libraries and electronic commerce. We note that most of the popular applications on the Internet are information flow-driven. This is not a coincidence, since the primary function of a network is to deliver information. Our goal, therefore, is to build software tools to support the development of scalable, composable, and predictable large-scale information flow-driven applications.

B. Concept of Self-Regulating Software

Informal Introduction. We say that a software module is *self-regulating* if it is able to satisfy a set of explicitly defined Quality of Service (QoS) requirements on its input and output, usually under certain environmental conditions such as sufficient resource availability. For example, a self-regulating multimedia streamer will be able to supply a video stream with certain bandwidth, latency, and jitter requirements, so the client can show the video with expected visual quality. The QoS requirements are called a *QoS Box* (QB, pronounced as "cube"). We do not require the self-regulating software to guarantee QB specifications, since that would imply advance reservations of all resources that may be needed. Instead of an absolute guarantee, self-regulating software notifies the system (and application) when it anticipates failing to meet the QB specifications due to any reason, usually because of insufficient resources.

The Hypothesis. Although relatively simple, we believe that self-regulating software will help application designers and programmers develop scalable, composable, and predictable large-scale information flow-driven applications. This belief comes from the following informal arguments about distributed software challenges. In the next section we will discuss the technical approach to make self-regulating software a reality.

Predictability Challenge. Using a light language, self-regulating software is software that runs well until it fails. By running well we mean achieving the QB specifications, and usually it fails because of insufficient resources. But *before* it fails, it tells the system to replace itself with another self-regulating software that runs well under the new

circumstances. Although there is no absolute guarantee, this is much better than blind best effort. The application always knows what is the result to be expected (from the currently effective QB).

Scalability Challenge. As we will explain in the next section, self-regulating software runs under a progress-based scheduling policy. In other words, self-regulating software receives just enough resources (if available) to make sufficient progress according to its QB specification. (Hence the name self-regulating.) Consequently, it is able to run under a variety of environmental conditions as well as on different system configurations.

Composability Challenge. We will adopt existing software technology for functionality composition (e.g., object-oriented software and Aspect-Oriented Programming). The difficult part of distributed software composition is with global properties such as end-to-end QoS requirements. The composite self-regulating software will run under a composite QB, where each component QB remains predictable, as well as the overall QB. Accordingly, each component self-regulating software makes its own progress, as well as the composite self-regulating software. If the composite QB cannot be satisfied by adding up the requirements from each QB, then the global (composite) requirement prevails. In the previous example of packet going from Atlanta to Seattle, one way to adapt to the global requirement would cause the component requirement (Atlanta to Chicago and Chicago to Seattle) to drop to 50ms each and then jobs rescheduled. The bottleneck between Chicago and Seattle would be discovered quickly.

C. Technical Approach

C.1 QB Specifications and Composition

QoS Properties. The QB requirements are defined broadly. For example, bandwidth, latency, and jitter are important performance properties for multimedia applications. Availability, security, and consistency are other examples of QoS properties important for large distributed applications. More concretely, a QoS property definition usually has three parts: (1) a specification of what the property should be, (2) predefined limits of that property, if any, and (3) the actual, current reading of that property. For example, an information flow may have a specified bandwidth of 1Mbit per second (specification), on a network with T1 (1.5Mbps) maximum bandwidth, and data currently flowing through it at 1.1Mbps (actual reading).

QBL. We will design a domain-specific declarative language (QBL) to describe the QB specifications. We will implement interpreters and compilers for QBL, as well as run-time environments. QBL will be extensible and incrementally implemented, adding more QoS dimensions as the project progresses. We will start with the QoS properties of streaming multimedia and move into real applications such as Environmental Observation and Forecast Systems (an OGI project of which Walpole is part) and Severe Weather Management Systems (a Georgia Tech project of which Schwan is part).

Composition of QBs. A composite information flow contains several information flow segments, each with its QB. For the composite information flow to be considered self-regulating, it will have a composite QB that describes the end-to-end information flow QoS requirements. In the conservative case, the system is self-regulating if the QBs at all levels are satisfied through the execution (in the sense outlined in Section B). In practice, there may be tradeoffs or conditional alternatives that relax the requirement on all QBs. We will support explicit disjunction on top of the implicit conjunction. For example, the composite QB may be satisfied if one of the several information flow paths achieve 1Mbps bandwidth.

C.2 Progress-Based Scheduling

Self-Regulating Schedulers. At the core of self-regulating software systems are the self-regulating resource managers. We will use feedback both as conceptual tool and software tool in the implementation of system support for self-regulating software. We borrow feedback concepts and techniques from control systems to reason about predictable adaptation. There are several advantages in using feedback. First, we will use feedback as a uniform and unifying approach to the management of all system resources, including CPU, memory, disk, and network. This is discussed in this section. Second, control systems already have a set of mathematical analysis tools for feedback systems that we can use as a starting point in the construction of a rigorous foundation for self-regulating software (see Section C.3). Third, feedback components have well-defined behavior that maps well into QB monitoring. The predictable behavior is also very useful during automated composition of information flows.

Progress-Based Scheduling. The basic idea of progress-based scheduling is to measure a job's progress towards its completion and allocate resources according to the observed progress. If the job is making insufficient progress, then more resources are allocated for it to catch up and finish on time. More precisely, we can allocate just the right amount of resources so the job makes forward progress according to a specified rate. This is the case of information

flow applications with QoS. The emphasis on on-time job completion is consistent with our goal to satisfy QoS requirements. This is very different from batch scheduling work where the goal is to maximize resource utilization.

Progress-Based CPU Scheduling. Software feedback is a natural mechanism to implement progress-based schedulers. A good example is our work on proportional share CPU scheduling [OSDI'99], where we implemented a scheduler that gives a specified proportion of the CPU to each job in the Linux kernel. We applied progress-based scheduling to a pipeline of processes connected by an information flow. By observing the buffer levels between processes, we measure the progress of each process. For example, if a buffer is full, the consumer is too slow and producer too fast. We then increase the proportion allocated to the consumer process and decrease the proportion of the producer. This kind of scheduler can satisfy the QB specifications of *real-rate* applications, where the information flow is produced (or consumed) by events external to the system (and outside the system control). Examples of real-rate applications include the handling of real world events such as sensor information capture (where real-time data may drop on the floor - permanently) and QoS specifications such as video display frame rates (where a movie may freeze when frames fail to arrive on time).

Network Bandwidth Scheduling. We have applied the progress-based scheduling idea to other systems resources. In particular, we have been studying network bandwidth scheduling [MMCN'01]. An example of useful application of progress-based scheduling is TCP-friendly network support for QoS. Due to the immediate exponential back-off of TCP under congestion, any protocol that is consistently aggressive (e.g., a slightly slower back-off) can eventually occupy all available bandwidth and drive down network bandwidth available to all TCP connections to zero. Progress-based bandwidth scheduler throttles the back-off time of protocols supporting QoS, by giving them the right proportion of the network bandwidth. Once the appropriate proportion is achieved, the QoS-supporting protocols start to backoff at the same time as TCP connections. This way, protocols that support QoS get their proper share of bandwidth and leave the rest to TCP. This is a highly desirable requirement for Internet applications.

Managing All Resources. The same way we have applied progress-based scheduling to CPU and network bandwidth, we will build progress-based schedulers for memory and disk, and eventually, manage all resources in the system this way. Our goal is to build an integrated scheduling system that is self-regulating with respect to each resource. Using the same mechanism (software feedback) for all resources, we simplify the reasoning and property verification of the scheduling system, its implementation and evaluation.

C.3 Rigorous Analysis of Software Feedback

The challenge of composition and predictability. Control theory informs us that composition of multiple well-behaved (QoS) and stable dynamic subsystems may result in a poorly behaved, and perhaps even unstable, composite system. It may have oscillatory behavior that does not settle down in a reasonable time, may converge to the wrong behavior, or may go to an extreme limit (e.g., no resources allocated to a job). We do modeling and controller design to avoid designing bad composite systems. It is easy to design subsystems that work well on their own, but their composition may not work as intended.

Composite system model from component models. A fundamental tool in classical (linear) control theory is called "block diagram analysis", in which a "closed-loop" (composite) system model is derived from the component models and system interconnections. This may be a bit harder in nonlinear and hybrid systems, but is necessary in order to do analysis or formal design.

Design of predictable, stable composite systems. Design methods, available in the control literature, assure predictable, stable composite systems under various conditions. There are various optimization methods, including tools for guaranteeing bounds if disturbances and inputs are bounded.

C.4 Software Tools

System Components. At the system level, we will build system components to support the execution of self-regulating software. Examples of critical system components include the progress-based schedulers outlined in Section C.2 and TCP-friendly streaming protocols that support QoS outlined in Section C.3. Other examples include software libraries such as the SWIFT software feedback toolkit, which provides the building blocks for the construction of simple and efficient feedback mechanisms for middleware and kernel level software. The systems level tools provide half of the *scalability* part and most of the *predictability* part of this research.

Programming Tools. At the software and programming level, we will adapt and build programming tools to support the development of self-regulating software. Programming techniques important for self-regulating software include specialization through partial evaluation and domain specific languages (DSL). An example of

program specializers is Tempo-C, which uses partial evaluation to eliminate unnecessary code automatically from C programs. (We also have a Java specializer.) An example of DSL tools is the transformation of a DSL interpreter into compiler through program specialization. Our work on QB specification and composition will leverage on DSL techniques and tools. For example, we will create and maintain a library of QB specifications that can be reused by our tools. The programming level tools provide the other half of *scalability* and most of the *composability* part of this research.

C.5 Large Technical Barriers

From past experience we know that building scalable, composable, and predictable distributed systems is a difficult problem. Space constraints prevent us from listing detailed technical challenges. We will discuss primarily large technical barriers that include many concrete technical challenges. The most obvious technical barrier is the combination of techniques that form Self-Regulating Software: software feedback, program specialization, DSL, information flow applications). To the best of our knowledge, this combination (and this kind of technology combination) has not been attempted due to the different kinds of expertise required. Our team has worked together successfully on pair-wise combinations and we have reasonable confidence that the grand combination to achieve Self-Regulating Software will work.

In addition, although each one of these techniques has been maturing, they still need further development so they can work together. A concrete example is the composition of QB and guarding of each component QB. Consider the example of end-to-end QB specification for the same latency as components. This requires adaptation of component QBs to "squeeze" their allowed latency, so the overall latency can fit. This is a dynamic optimization problem with changing constraints, so only self-regulating adjustments will work. Another example is the rigorous analysis of software feedback. It is difficult to apply control systems analysis techniques directly if self-regulating systems are dynamically changing their models, the number and nature of their components.

C.6 Practical Impact

Despite the success of 3-tier client/server systems (e.g., online shopping systems), the development and maintenance of large distributed applications have been limited by the Scalability, Composability, and Predictability challenges. For example, the current generation of price comparison shopping agents tends to collect price information from several sources in an N-tier configuration. It is not uncommon for these comparison agents to take up to a minute to respond. To overcome this kind of unbounded latency, a self-regulating agent can specify a composite QB with global latency of 10 seconds, and the component QBs will attempt to adjust to this global limitation. Consequently, the agent will be able to provide a consistent answer after about 10 seconds with the information from the sources able to make it on time.

Of larger impact will be the next generation information appliances in an ubiquitous computing environment. Consider a Personal Guide PDA, which downloads maps of the neighborhood as you travel, and offers, maps and travel tips automatically. Such a Personal Guide will make getting lost a concern of the past, saving lives by telling people how to get out of mountains, and saving time and money by telling people how to get around traffic in cities. The main roadblock towards this vision is efficient information flow to the device, since we already have the hardware (for both client and server) as well as the information in databases. In terms of military relevance, effective personal guidance is clearly valuable, particularly in dynamic environments where fresh information about friend and foe can mean the difference between success and failure of a mission. Other important applications include real-time decision support and dynamic mission re-planning (e.g., missions to capture or destroy moving targets such as mobile missile launchers or fugitive terrorists).

D. Collaborators

This research is in collaboration with Prof. Ling Liu (adaptive data management) and Prof. Karsten Schwan (adaptive systems software) at Georgia Institute of Technology, Prof. Jonathan Walpole (adaptive resource management) at Oregon Graduate Institute of Oregon Health and Sciences University, Prof. Molly Shor (formal analysis of software feedback) of Oregon State University, and Prof. Charles Consel (program specialization and domain specific languages) of University of Bordeaux (France). Application area collaborators include Prof. Antonio Baptista (environmental monitoring and forecasting systems) at Oregon Graduate Institute and Prof. J.C. Lu (manufacturing and supply chain management) at Georgia Institute of Technology.