

Autonomic Computer Systems CS321: Motivation, What is “autonomic”?, Self-Star, Control Loops

September 20, 2011

Christian Tschudin, Manolis Sifalakis (and C. Jelger, L. Yamamoto)
Departement Mathematik und Informatik, Universität Basel

Overview 2011-09-20

- Motivation – about Faults, Errors and Failures
- IBM's Initiative
- Self- {configuration — optimization — healing — protection }
- Control loops
- Policies

Why do systems fail? E.g. hardware component

- ENIAC computer (1946) had 17'500 tubes
- Myth that failure rate was too high, such that a tube would fail before a computation was done
- Reality: 1 tube every 2nd day, locatable in 15 minutes.
- Initially, it was several tubes a day, but then they did not switch the computer off

Today: hardware has become incredibly reliable.

Why do systems fail? (hardware contd)

MTBF - Mean Time Between Failure

(how long it takes on average before a component fails)

- Light bulb: 2'000h
LED light: 100'000h
- Harddisk (1990): 200'000h
Harddisk (2005): 1'200'000h

1.2 Mh MTBF sounds like a lot,
until you divide by the 50'000 PCs
in your company: 1 failure every day!

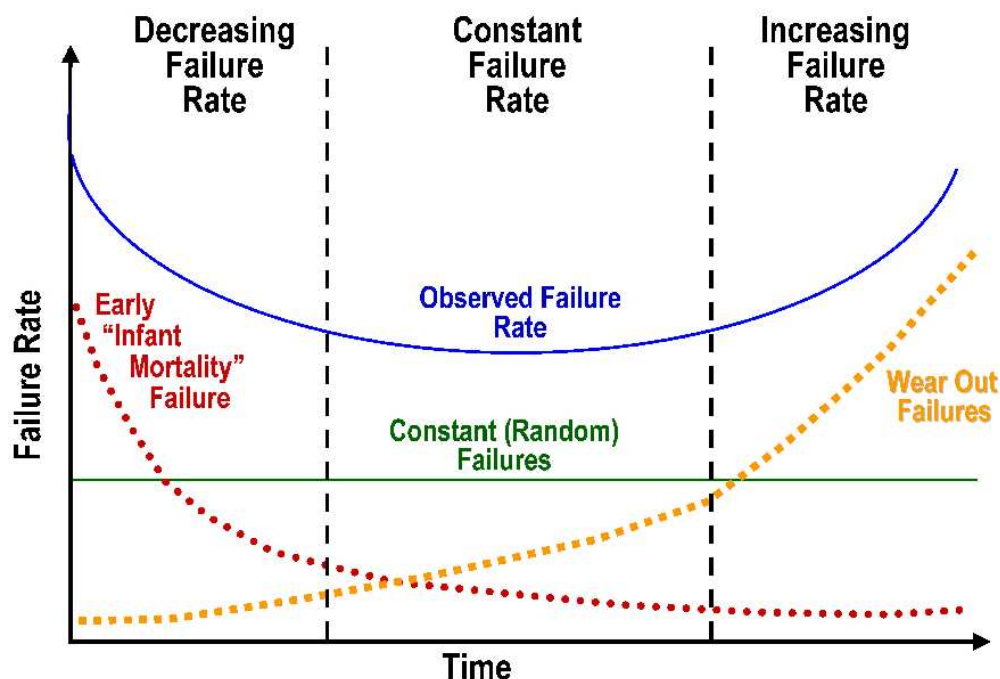
Why do systems fail? Complete PC systems

Gartner group, June 2006, “Annual PC Failure Rates Improving”

<i>purchased in</i>	2003-2004	2005-2006
Desktops:		
year 1	7 %	5 %
year 4 (projected)	15 %	12 %
Laptops:		
year 1	20 %	15 %
year 4 (projected)	28 %	22 %

Average Annualized Failure Rates for PCs

“Bathtub” model



Coping with Failures

Bathtub model:

- “Burn in”: manufacturer runs newly made devices for hours at high temperature and over-voltage, this cuts away the left peak for owner.
- Early retirement:
the owner does preventive replacement before too many devices start to fail.

Bathtub model also for software? Partly yes!
Infant mortality also observed

General Attitude Towards Failures

Embrace failures!

- There is no perfect system, hardware and software:
Failures must be managed.
- Failures are not binary:
 - transient failures
 - partial failures, tolerable failures etc
- Failure **containment** techniques → field of “fault tolerance”

Terminology: Fault, Error, Failure

in German: “Fehler” is too broad.

- Fault (Defekt):
physical, design or software flaw
- Error (Fehler):
incorrect behavior caused by a fault
- Failure (Ausfall, Versagen):
inability to perform according to spec because of an error

often a chain:

fault → **error** → **failure** → **fault** → **error** ...

Dealing with Faults

Goal: **breaking the fault/error/failure chain**, fault containment

Four different strategies (for HW and SW):

- fault **prevention**: preventing occurrence or introduction,
e.g. type-safe languages, good SW engineering practices
- fault **removal**: process to identify and remove faults, e.g. testing
- fault **tolerance**: delivering correct service in the presence of faults
- fault **prediction**: estimate future incidence and likely consequence of a fault

Fault Tolerance

Implemented by *error detection* and subsequent *system recovery*.

- Error detection important, not always easy
- Recovery has two activities:
 - error handling (undoing the harm)
 - fault handling (prevent from faulting again)
- Three forms of fault handling (see also DB, NW course):
 - rollback (to a checkpoint)
 - rollforward (progress to a state without error)
 - compensation (error elimination, e.g. in FEC)

Preview to Autonomics

- Error detection ...
becomes “self-diagnosis”
- Fault handling ...
becomes “self-healing”

Error Detect./Fault Handling Example: Airbus 320



All actuators (flaps, engines etc) controlled by computer. Fly-by-joystick, commands via computer network.

- 5 central flight computers:
 - hardware variety (68010, 80186)
 - software variety (different software houses)
 - separate error checking and debugging
 - fault tolerance through majority voting among computers!

Redundancy!

Fault tolerance relies on several forms of redundancy for being able to recover:

- *Hardware* redundancy:
execute in 2nd path
- *Time* redundancy:
execute a 2nd time
- *Information* redundancy:
error correcting codes
- *Software* redundancy:
SW copies (replicas, alternate progs, consistency checks)

Shift Focus from Hardware to Software

- up to 1990ies: hardware failures dominated
- since then: software faults and limitations win
 - better HW production processes
 - new HW technology, hot swap etc
 - SW: more complex tool chains
 - more difficult to handle (fault prevention)
 - more difficult to detect
 - software-down-time still here (usually not a hot-swap)
- and then: human (operational) failures!
Important to include “human failure mode”

Reliability and Availability

- Reliability $R(t)$: probability that the system performs as specified without interruption over the entire interval $[0,t]$
- Availability $A(t)$: probability that the system is up and running correctly at (any) time t
(note: not the same as $R(T)$ which only looks at $[0,t]$)

We are interested in availability. If a component is not reliable, but can be fixed in very short time when it fails, then we have high availability, overall.

Availability

- MTTF – mean-time-to-failure (from manufacturer)
- MTTR – mean-time-to-repair (process, possibly human)

$$A = \frac{MTTF}{MTTF + MTTR}$$

- MTBF – mean-time-between-failure:

$$MTBF = MTTF + MTTR$$

which is almost the same as MTTF

Availability (contd)

- $A(t)$ usually written as series of nines: 99%, 99.9% etc
- Magic “five nines”: 99.999%

Availability (uptime)	Unavailability (downtime, min/year)	Availability Class
90.%	50000	1
99.%	5000	2
99.9%	500	3
99.99%	50	4
99.999%	5	5
99.9999%	.5	6
99.99999%	.05	7

Economic Relevance of Availability

Cost of Downtime [Marcus Stern 2000]

Industry	Business Operation	Average Downtime Cost Per Hour
Financial	Brokerage operations	\$6.45M
Financial	Credit card/sales authorization	\$2.6M
Media	Pay per view	\$150K
Retail	Home shopping	\$113K
Retail	Home catalog sales	\$90K
Transportation	Airline reservation	\$89.5K

Overview 2011-09-20

- Motivation – about Faults, Errors and Failures
- **IBM's Initiative**
- Self- {configuration — optimization — healing — protection }
- Control loops
- Policies

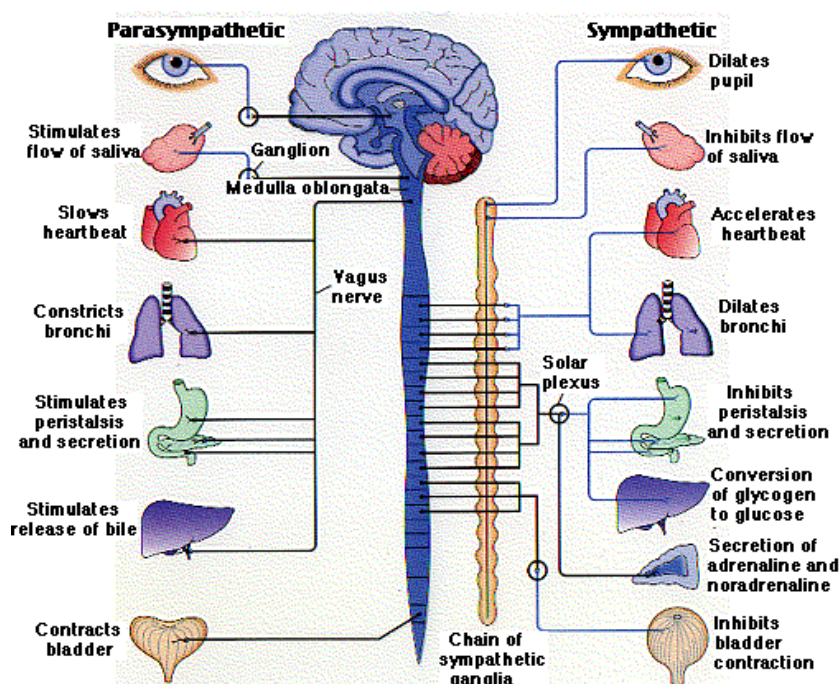
Why Autonomics?

“Civilization advances by extending the number of important operations which we can perform without thinking about them.”

– Alfred North Whitehead

- “Autonomic Computing” coined 2001 by IBM, which had become a service company
- Motivation: Proliferation of computer devices has led to unprecedented levels of complexity, how to train and maintain?
- Mission: “Build computer systems that regulate themselves much in the same way our autonomic nervous system regulates and protects our bodies.”

Autonomic Nervous System



Self-Properties (for Autonomic Computing)

8 or 4 properties, depending on where you look:

- self-configuration
self-optimization
self-healing
self-protection
- plus: self-aware, self-learning, operate in heterogeneous computing environment, anticipate and adapt to user needs.

In the literature, from others: many more “self-star properties”:
self-organizing, self-adaption, self-management, self-deployment,
...

Four Elements of IBM's Autonomic Computing (CHOP)

- Self-**C**onfiguring
e.g., automatic test and installation of SW releases
- Self- **O**ptimizing
e.g., out-source number crunching during day time
- Self-**H**ealing
e.g., system can restart applications if they fail
- Self-**P**rotecting
e.g., update patches, active intrusion detection

More on these self-star further down.

An Even Bigger Picture

Human is the “weakest link” in the way of IT progress:

- Today: “babysitting” of complex systems, humans are “in the (control) loop”.
- Goal 1: Get human *operators* “out of the loop”.
Usual politically correct version: put operators “on the loop” instead of “in the loop”.
- Goal 2: Get human *engineers* “out of the loop”:
even searching for solutions will become too complex.

IBM's Vision, Manifesto in 2001

“Grand Challenge of building computing systems that regulate themselves.”

- Manual-user admin, maintenance, management cannot keep pace with rate of infrastructure change.
- Managing individual HW or SW modules does not guarantee overall performance.

In numbers:

1 dollar spent on computing infrastructure →
10 dollars spent on management.

Automatic vs Autonomic

- Automatic: pre-programmed task execution
 - system works fine until something goes wrong
 - at latest now, human intervention is needed
- Autonomic: self-regulation
 - system response is also automatic, but modulated
 - system can compensate or work around problems
 - no human intervention needed

Both need engineering effort, though.

Example: Install set of new SW upgrades

- Assumption:
 - means exist to *automatically* deploy upgrades
 - *automatic* regression tests exist
 - *automatic* problem determination
- Autonomic approach
 - deploy, run, testplus, in case of problems:
 - revert
 - identify problematic component, isolate it
 - re-start with the reduced set of updates

Update Example Contd

Potential application: building a new Linux/FreeBSD/etc release

- Programmers submit their patches, new applications
- System automatically makes release candidate:
(regression tests)
 - does kernel still compile
 - can all library dependencies be resolved?
 - do unit tests still work?
- Predicting release dates:
 - analyzing the ratio of bug fixed to bugs introduced

Overview 2011-09-20

- Motivation – about Faults, Errors and Failures
- IBM's Initiative
- **Self-** {configuration — optimization — healing — protection }
- Control loops
- Policies

Element 1: self-configuration

IBM's concern: computing, hence the following wish list

- automatic SW
 - deploy
 - install
 - configure
 - re-configure
 - config documentation
- while adhering to administrative directives, and producing reports on compliance

Self-configuration (contd)

Self-Configuration in many other contexts:

- Networking:
 - addresses, routes, zero-config
- Parallel Computing:
 - assigning free processors
- (software) Multi-Agent Systems, MAS:
 - dynamic agent allocation and scheduling
- Complex systems:
 - self-organization (e.g., find new spacial configurations)

Self-Configuration (contd 2)

Definition for our usage:

- **system is self-configuring**



the system is able to (re-) configure itself according to high-level policies.

This is not necessarily autonomic by itself, but is required for an autonomic system

Self-Configuration (contd 3)

Observation on current status (regarding configuration):

- Data centers have
 - multiple vendors, platforms, software systems
- Installation/configuration/integration of new elements (HW, SW, policies) is
 - time consuming (days, weeks)
 - error prone

Goal: Automated config of components according to high-level policies **and** then rest of system adjusts automatically.

Self-Configuration:

- Examples mentioned before:
software updates, regression tests

Overview 2011-09-20

- Motivation – about Faults, Errors and Failures
- IBM's Initiative
- **Self-** {configuration — optimization — healing — protection }
- Control loops
- Policies

Element 2: Self-Optimization

- Observation: Huge number of tuning parameters, for HW and SW
 - cache size
 - timeout values
 - cpu and bandwidth allocation
 - dimensioning of internal data structures, hash tables
 - several algorithms for same task, different profiles

IBM's description, also our definition:

system is self-optimizing \Leftrightarrow the system and its components continually seek to improve their performance and efficiency.

Self-Optimization (contd)

How far can self-optimization go?

Many places to optimize:

- Design time:
Software architecture
- Implementation time:
choice of algorithms, language, optimizing compiler
- Run time (e.g. TCP adaption)

Current understanding: only parameter setting at run-time.
(adaption vs evolution, see later on)

IBM's Autonomic Elements, practically: Self-Opt

Self-Optimization examples:

- adding more (computing) resources on demand, when some apps become too slow
- automatic outsourcing of tasks (Grid, Cloud Computing)
- picking optimal connectivity with sub-second response time (WLAN, ethernet, GPRS ...), or cheapest route etc

Optimization includes resolution of conflicting goals.

Overview 2011-09-20

- Motivation – about Faults, Errors and Failures
- IBM's Initiative
- **Self-** {configuration — optimization — healing — protection }
- Control loops
- Policies

Element 3: Self-Healing

Problem solving: an analytic activity for humans only?

Wish list:

- reactive:
 - recover from events causing failure or malfunction
- proactive:
 - anticipate/predict failures

Problem solving includes:

find cause, find cure, test cure, deploy

Self-Healing (contd)

IBM has a somehow restricted definition:

system is self-healing \Leftrightarrow

the system automatically detects, diagnoses, and repairs localized software and hardware problems.

- Mostly centered around configuration errors,
see regression test for SW upgrades
- Mostly based on log analysis
- But also requires infrastructure support
(moving a set of apps and their data to a new server)

IBM's Autonomic Elements, practically: Self-Heal

Self-Healing:

- “self-correcting job control language”:
job failures are analyzed, job is automatically restarted
- Database index file is corrupted:
automatic re-indexing, testing, going productive
- Hardware failure, server goes down:
“[websphere app server v6] can transfer those transactions to another server [automatically]”

Self-Healing (contd 2): as an additive feature

- Characterizing IBM's approach: *adding* self-healing features
Self-healing mostly applies to configuration parameters:
 - which server, which update etc
- Next level (beyond IBM): self-healing inside the apps
 - what algorithm to use
 - what data structure, what encoding etcRequires trade-offs and (internal) choice:
 - this algo is more robust, but slower

Still, this is “parametrization”.

Self-Healing (contd 3): Deep self-healing

Our definition:

system is self-healing \Leftrightarrow

the system asserts *goal integrity* over long times.

- Note: Code can change, changing the “performance envelope”!
- Self-healing at the code level:
 - app controls its own code, not only parameters
 - can walk algorithm space, not just config space
- First step:
 - assert code integrity over long times
 - example: application fights viruses itself

Overview 2011-09-20

- Motivation – about Faults, Errors and Failures
- IBM’s Initiative
- **Self-** {configuration — optimization — healing — protection }
- Control loops
- Policies

Self-Protection

According to IBM:

system is self-protecting \Leftrightarrow

the system defends against malicious attacks or cascading failures, can prevent systemwide failures.

- Ambitious and difficult task:
attacks are not known in advance
- Focus on automatic reaction, tiered security
(breach on one level does not endanger whole system)

IBM's Autonomic Elements, practically: Self-Prot

Self-Protection:

- Confirm ability to backup and recover data resources
- Network monitoring and IDS, automatic disconnection of suspicious computers
- Verify that all client machines have latest patches
- Track security advisories

- Motivation – about Faults, Errors and Failures
- IBM's Initiative
- Self- {configuration — optimization — healing — protection }
- **Control loops**
- Policies

IBM's Proposed Methodology: System Decomposition

Global self-* behavior cannot be achieved “as a whole”

- Autonomic system is broken into smaller but autonomic entities, each being dedicated to fulfill a specific task.
- Reduces complexity of designing a large system
- Potentially increases the complexity of managing the resulting multiple interactions.

The self-* behavior is an “external” characteristics, i.e. the smaller entities that compose the system do not necessarily exhibit self-* properties.

“Autonomic Element”

Autonomic Element = “manager” for some resource or task

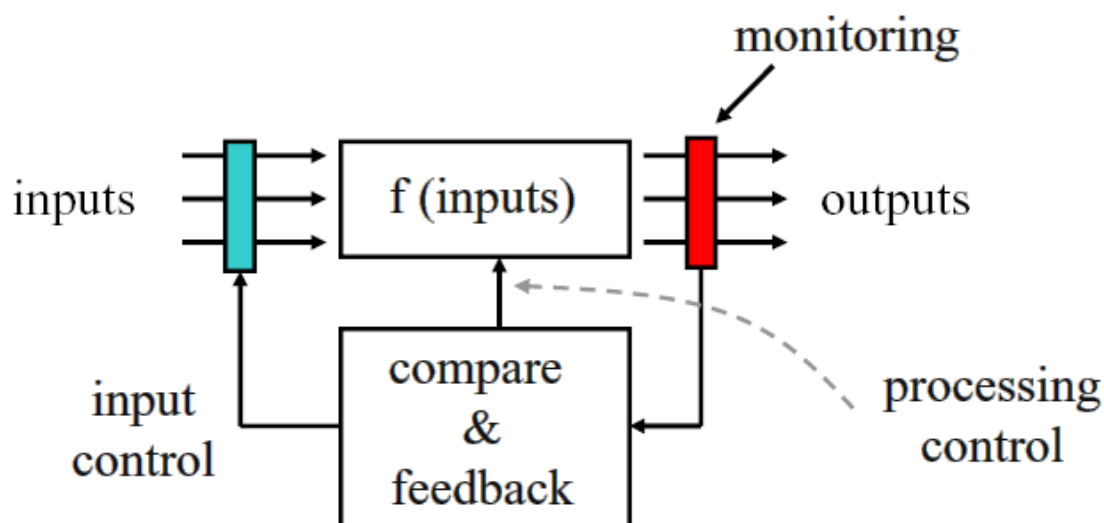
- Has defined behavior, constrained by policies and goals.
- Decisions based on observations of their environment, policies and goals, and “configuration-range”.
- Negotiation between autonomic elements

Examples:

- Install or upgrade software.
- Restart a system or component after a failure.
- Isolate systems after intrusion detection.
- Re-configure task to cope with changing conditions (e.g. load)

Input/Output Model, Control Loop

“Steering” of autonomic elements is achieved via control loops



Input can vary, self-adjustment between actual and desired output.

Autonomic Ctrl Loop

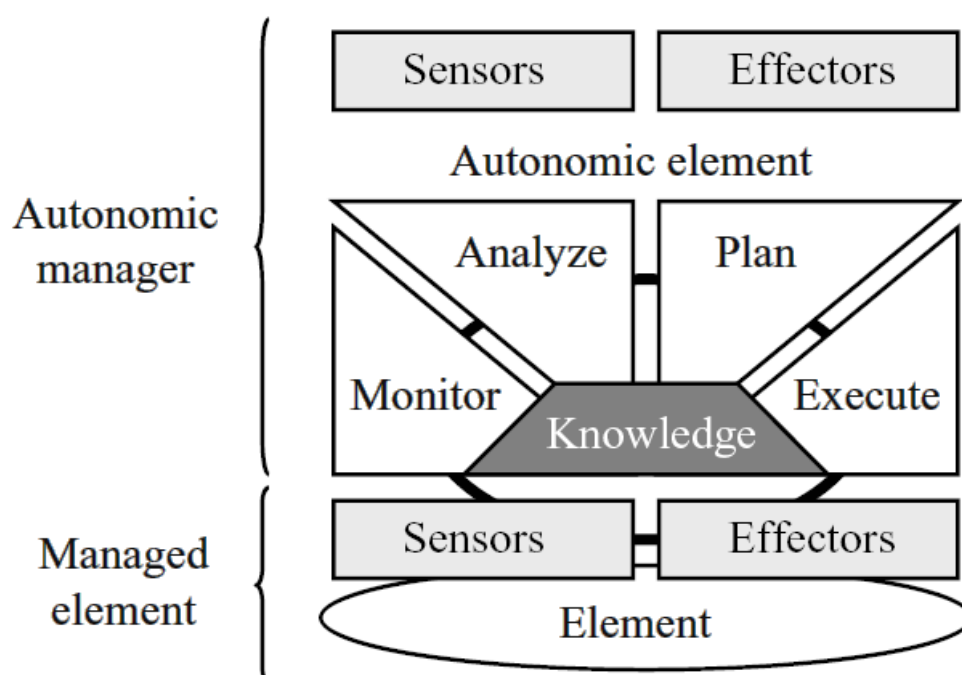
“Intelligent” control inside an autonomic element, based on four phases/activities:

1. Monitor: collect and filter data.
2. Analyze: compare system operation wrt policies and goals.
3. Execute: flexible and re-configurable exec of tasks
4. Plan: (re-)configure operation with respect to policies and goals.

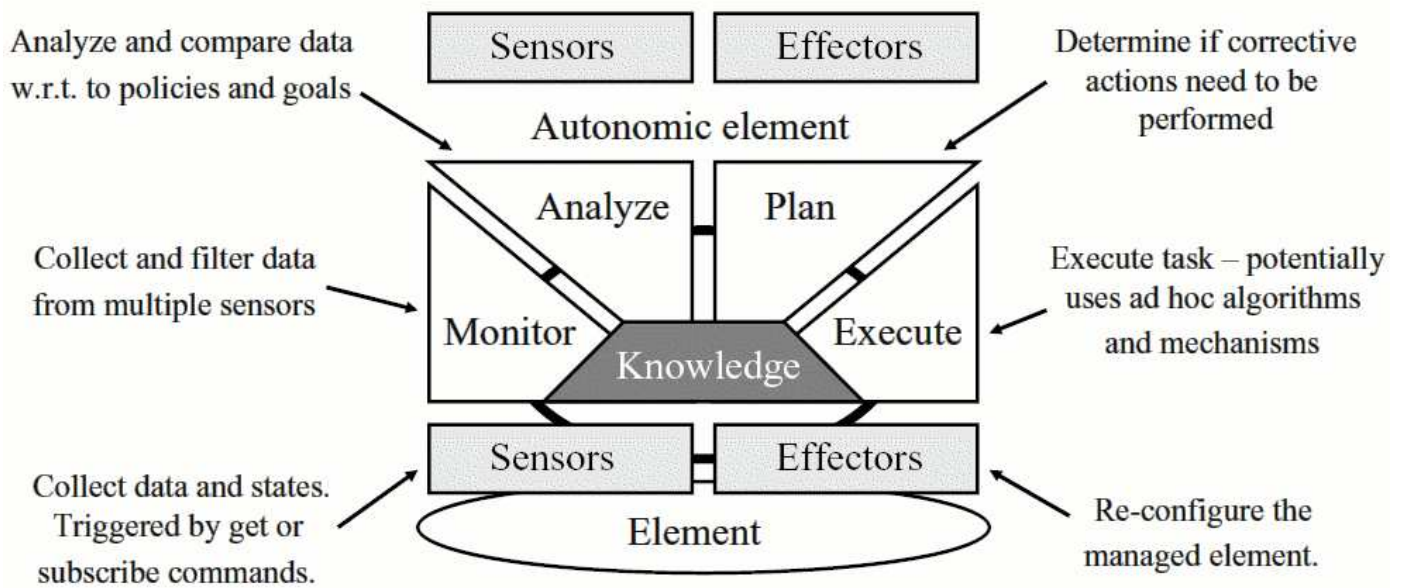
Needs Sensors: to collect data, states of the managed element.

Needs Effectors: to apply changes (i.e. re-configure)

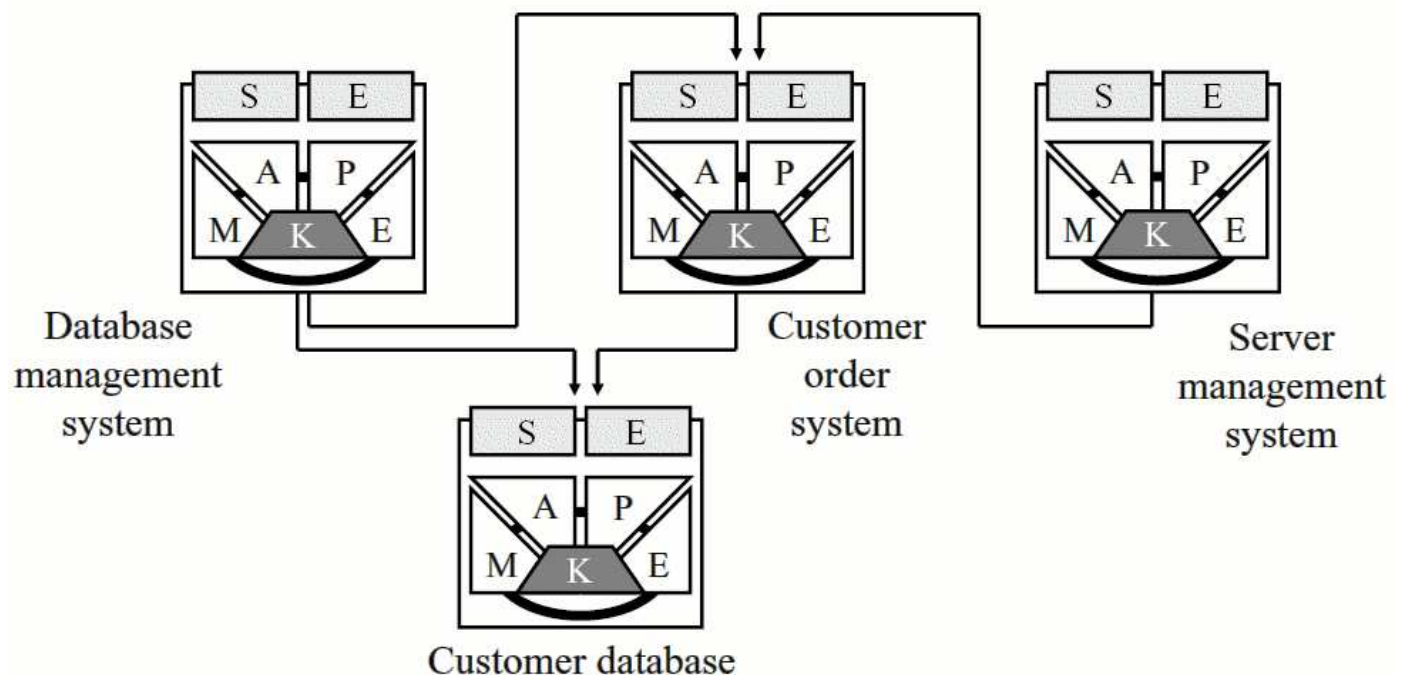
“Intelligent Control Loop”



“Intelligent Control Loop” (contd)



Nested Ctrl Loops: Managed Managers



- Motivation – about Faults, Errors and Failures
- IBM's Initiative
- Self- {configuration — optimization — healing — protection }
- Control loops
- **Policies**

Policies

System (or ctrl loop) cannot guess “what is good behavior”:

- Autonomic element attempts to fulfill its goals according to *policies*
- Wrong goals + wrong policies = wrong outcome
- Challenge: deriving consistent rules for multiple autonomic elements such that global behavior is fulfilled.

Examples of policies:

- Minimize memory and disk usage for database.
- Maximize lookup speed inside database.
- What about “minimize mem usage and maximize speed”?

Control Loop: Plan Component

Plan component re-configures system behavior based on analysis of current behavior.

- Adaption range might be pre-defined
 - e.g. Adaptability of TCP is large but limited by design.
- Might consider only current state (TCP has little memory)
- For better adaptivity: Knowledge of the past and “learning from experience” are required
- System learns the (good or bad) consequences of actions taken (TCP could “invent” new window management or congestion control).

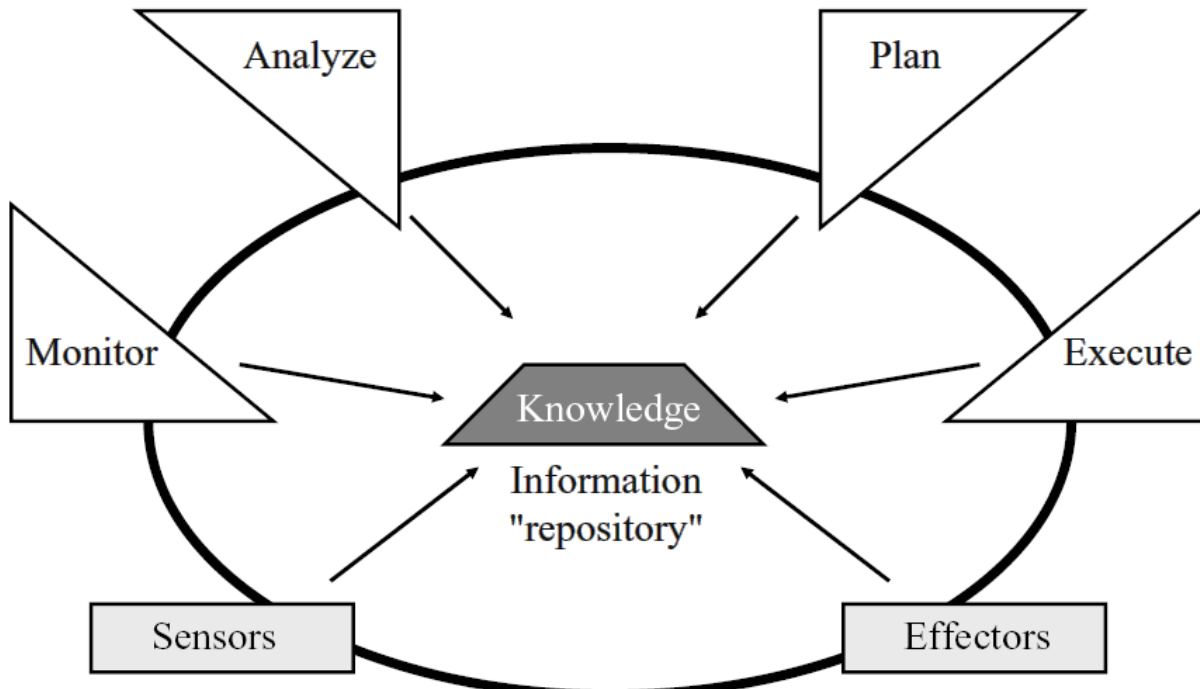
Control Loop: Analyze Component

Infers system behavior based on current and past observations, and according to current policies and goals of the system.

- May have to operate on incomplete data sets or partial view of the running elements.
- Can potentially request additional data from the monitoring element

Critical part of the system: even the best car-driving algorithm cannot get round a cliff if cameras don't look in the right direction at the right time.

Knowledge = information repository



Control Loop: Knowledge Component

“Database” storing data, events, states, collected by sensors, and decisions made by the autonomic element(s).

- Potentially maintains “a history” of past events and changes (e.g. to cope with long-term changes)
- Information should be stored in standard format, potentially in a distributed manner.
- Information sampling is dynamically adjusted according to load (CPU, bandwidth), storage space, system health
- Information might be compressed: averages, linear regression


Assessment of IBM's definitions/efforts

- No magic – it's the mind set!
Design for self-* in mind
- Additive approach:
 - add self-* “features”
- Autonomics with a methodology:
 - structure template for autonomic elements
 - loop model: plan, execute, monitor, analyze
 - loop sits “on” the managed element
 - controller hierarchy

Assessment of self-* properties in general

- Properties are not orthogonal
 - self-opt results in re-config
 - self-healing as special case of self-opt?
(minimize number of open problems)
- Properties are not specific about their target
 - design/compile/deploy/run-time
- Properties are not specific about system:
 - closed (static) set of elements
 - open system, distributed system, including evolution?

Extending the Landscape: What is Adaptive?



Evolutionary programming
(algorithm generation, genetic
algorithms, AI-based learning)

Algorithm selection

Generic or parametrized algorithms

Online algorithms (deterministic,
randomized, probabilistic)

Conditional expressions

(Oreizy et al, 1999)

Adaptive (contd)

- “Conditional expressions”:
 - reactive
 - predetermined fixed response
 - examples: network protocols
- “Online algorithms”:
 - assumption that future events are uncertain
 - keeping options open, anticipate problems
 - has to measure system state, reacts accordingly
 - example: swapping, opportunistic execution inside CPU

Adaptive (contd 2)

- “Generic/parametrized algorithms”
 - start an algo according to context, environment
 - example: C++ templates (adapt to data types)
- “Algorithm selection”
 - fixed set of implementations available
 - selection algorithm, or “adaption management”
 - example: optimizing compilers try several strategies
- “Evolutionary”
 - adaption to yet unknown border conditions
 - needs “evolution management” to keep system alive

Adaptive: Revisiting “Reliable Transport”

In the network lecture, we went through stages:
first handle bit errors, then message loss, then congestion, etc

Apply “adaptivity spectrum” to protocol variations:

- conditional expr: selective retransmission
- online algo: compute timeout value
- generic: choosing TCP buffer and window sizes
- algo selection: TCP vs STCP vs UDP vs ...
- evolution: transport protocol “breeding”

Outlook “Autonomic Computing”: Eternal SW Systems

Software decays like hardware. **Can we** write SW systems which can

- adapt to changing border conditions
- evolve on their own
- optimize their own code bases,
- control their own SW replication, as well as data replication?

Goal: SW life time in the range of a human life, not a few years.

Eternal SW Systems was in a 2007 EU call for research proposals.