SOME NOTES ABOUT

PROMISE THEORY

AND HOW TO APPLY IT TO SYSTEMS

Mark Burgess

2015

Contents

Chapter 1

Promise Method

Promise theory is not about data modelling (encapsulation), and it is not a form of network protocol. Promises are the opposite of requests and requirements.

- Promise theory is about modelling *causation, change and balance* between communicating agents (human or machine),
- It is about finding the necessary and sufficient conditions for cooperation between distributed agents.

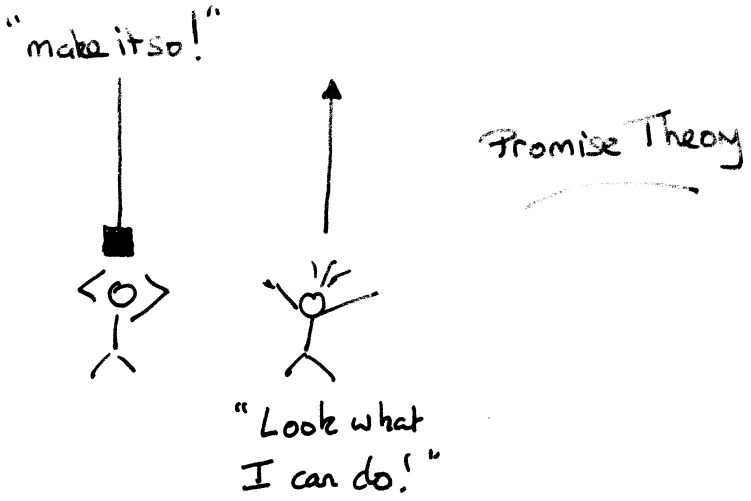

Figure 1.1: Promise theory is about turning in requirements and obligations into the "voluntary" promises of the appropriate agent.

The questions to always ask for every promise are:

- What agency/thing in the system is going to be responsible for making and keeping the promise? (Agents can only promise their own behaviour)
- What actions/operations repair a promise not kept? How are they parameterized and how do they converge to a desired state?

Promise Theory does not replace data modelling. It should be used in advance of of data modelling to construct small, minimal data models for autonomous, rather than large monolithic structures that cover all parts.

The benefits of a promise theory approach come from thinking about management through constraints, without and detailed operational protocols. The model of promises is very like the modern software notion of microservices.

1.1 Why do we need promises?

A promise communicates intended outcomes from one party to another. It is a *description* or *documentation* of intent: past, present or future. A promise is made by someone or something, to themselves or someone/something else. Promises thus describe intentional relationships, and form a network. However, a promise is not a network protocol.

Without understanding intended outcomes clearly, we can lose track of what is happening in a even the simplest of systems. Focusing on commands and interactions leads to a narrative about the user instead of a narrative about the system. Promises restore focus on design goals.

Promise Theory is not an agenda. It is not a design philosophy. It is not an information architecture. However, it can help us to make sense of all these things, be applying simple principles and tracing their consequences relative to a specific problem.

1.2 What a promise looks like

It's useful to have a formal way of talking about promises in technology. On paper, a promise is denoted like this:

$$\text{Thing} \xrightarrow{\text{promise description}} \text{Other} \tag{1.1}$$

The important details about this lie in how we formulate the *description*, i.e. what is the promise about? This is where data representations, domain specific languages, and even protocol specifications can be useful.

Whatever tools we decide to use, a promise can be represented in a number of ways, as language, document, by signalling, etc. The key thing to remember is that a promise communicates (somehow) intended outcomes from one party to another.

1.3 Autonomous agents make promises

All operational and passive 'things' in promise theory are called agents, e.g. devices, persons, software, etc. They possess 'agency' i.e. they exhibit behaviours and make decisions, whether intentionally or unintentionally, directly or on behalf of a person.

All intentions can ultimately be traced back to a person (at least until we meet other intelligences), however we build technology to capture that intent and channel it on our behalf. So, in technology, intent by proxy is very important.

One of the benefits of promise theory is to understand when and how to exploit autonomy, i.e. to understand when it is, or is not, enough to understand a system.

1.4 Impositions

Promises play an important role in autonomous cooperation, but not all intentional behaviour occurs spontaneously from within agents. We also need a way to model things like suggestions and requests, where one agent attempts to impose on another's autonomy to indicate a behaviour it desires. We call such communications *impositions*, as they challenge the autonomy of other agents in some respect.

A message intended to induce voluntary cooperation in another agent. We denote an imposition from agent A_1 to agent A_2 by the symbol (imagine a fist):

$$A_1 \xrightarrow{b}\blacksquare\ A_2 \tag{1.2}$$

Impositions have scope in the same way as promises.

Impositions come in varying degrees of strength, for example:

- Hints and suggestions.
- Requests and proposals

- Requirements and specifications.
- Commands and demands.

Impositions are not promises because they cannot be kept by the agent who makes them. Moreover, impositions are not quite obligations, as the latter imply some form of penalty for non-compliance. We may think of them as a desire to impart intentional behaviour in an external entity.

1.5 What is promise theory for?

Promises documenting relationships between autonomous components within a system, and describing their interactions. The goal is to determine whether all necessary information and behaviours are passed between the parts, in order to enable them to work together in keeping promises. Each agent is responsible for keeping its own promises, but together a collection of agents can achieve things that no single agent can achieve.

PT has a service oriented flavour. It is independent of any technology or implementation.

PT contrasts with command and control systems. A promise is not a command. It is a declaration of what is intended. Unlike a command, it persists and remains in an advisory capacity until it has expired.

1.6 Tenets of promise theory

The basic tenets of promise theory may thus be summarized as follows. We explain these in more detail in the following chapters.

1. Agents are autonomous. They can only make promises about their own behaviour. No other agent can impose a promise upon them.
2. Making a promise involves passing information to an observer, but not necessarily a message in the explicit sense of a linguistic communication[1].
3. The assessment of whether a promise is kept or not kept may be made independently by any agent in its scope.
4. The interpretation of a promise's intent may be made independently by any agent in its scope.
5. The internal workings of agents are assumed to be unknown. Knowledge of them may be assessed from the promises they make, and keep. However, we may choose the boundary of an agent wherever we please to hide or expose different levels of information, e.g. we may think of a car as an atomic vehicle, or as a collection of agents working together.

I'll shall refer to the following key concepts:

- *Intention*: The subject of a promise. Any agent can harbour intentions, whether directly or by proxy, e.g. to 'be red' for a light, or 'to win the race' for a person.
- *Promise*: When an intention is publicly declared, it becomes a promise. Thus a promise is a stated intention.
- *Imposition*: This is an attempt to induce cooperation in another agent, i.e. to implant an intention. It is complementary to the idea of a promise. Degrees of imposition include: hints, advice, suggestions, requests, commands, etc.
- *Obligation*: An imposition that implies a cost or penalty for non-compliance. It is more 'aggressive' than a mere imposition.
- *Assessment*: A decision about whether a promise has been kept or not. Every agent makes its own assessment about promises it is aware of. Often assessment involves the observation of other agents' behaviours.

1.6.1 Promises as 'voluntary cooperation'

When we give commands we have to assume that the commands will be understood, obeyed and be successful. This is an unreliable assumption. When we document what agents promise to do, rather than what we hope they will do, we directly document what each components has to do to satisfy what was intended. In a command model, the recipient of commands needs telling what to do. In a promise model, that agent is assumed to already know what to do (it doesn't need convincing or telling). Thus we start from what has already been understood.

Another way of thinking of a promise is as a measure of 'voluntary cooperation'. Instead of thinking that technoloy has to do what we tell it, we turn this upside down and take the point of view that technology only promises to exhibit certain behaviours. We can impose a desire, or promise to use a capability, but technology does not work because we ask it to. Either it works or it doesn't. If it doesn't work, this might be because

- It is unable to work.
- It is unwilling to work.

In either case, we say it is autonomous, meaning that we cannot force it.

Voluntary cooperation can be used as a very pragmatic engineering methodology. It forces us to deal with the idea that people and technology will not always do what we want. It forces us to realize that if we want something, it is up to us to obtain enough promises to secure a likely outcome.

1.6.2 Promises and trust belong together

The usefulness of a promise is intimately connected with our *trust* in the agents that make promises, and their ability to keep the promises. This relates to reliability.

Equivalently we can talk of the *belief* that promises are valid, in whatever meaning we choose to apply. In a world without trust, promises would be completely ineffective.

1.6.3 Promise proposals and signing

It is possible to propose a promise without making it. This is how we make contracts and terms and conditions. A promise proposal is a statement of a promise that is posited for consideration by one or more parties, prior to keeping or discarding the promise. In other words, a proposal is a complete description of a possible promise, with the caveat that this is not yet intended. Promise proposals are often discussed as part of treaty negotiations and commercial relationships, such as contractual relationships.

1.6.4 Promise polarity

Visualizing promises in terms of graphs or networks helps us to see even implicit channels of influence explicitly. However, it requires some care, as the channels of influence follow the scope of promises rather than just the obvious promiser to promisee route.

It is helpful to denote the flow of intent using signed promises. We shall denote promises with a $\pm$ signs, as below:

$$A_1 \xrightarrow{+b} A_2 \ \ (\text{I will give } b) \tag{1.3}$$

$$A_1 \xrightarrow{-b} A_2 \ \ (\text{I will accept } b) \tag{1.4}$$

We shall sometimes emphasize the promise to use another agent's promise of a service by using a more visible notation:

$$\begin{aligned} A_1 \xrightarrow{S} A_2 \ \ \text{or} \ \ A_1 \xrightarrow{+S} A_2 \\ A_2 \xrightarrow{U(S)} A_1 \ \ \text{or} \ \ A_2 \xrightarrow{-S} A_1 \end{aligned} \tag{1.5}$$

1.6.5 Idempotence in repetition of promises

What if we repeat a promise to pay 100 Euros to someone; have we kept our promise if we pay 100 or 200 Euros?

Rule 1 (Separate events have separate types) *Each new labelled promise that is fulfilled by the same type of action, must be labelled by a formally different type promise.*

Certain promise types are therefore related, or we can think of them as being parameterized. For example, the different payments above have different types, otherwise we lose track of total value. e.g. τ_1 is payment 1, τ_2 is payment 2.

Rule 2 (Idempotence of promises) *The promise combination operator is* idempotent*, so repeating the same promise does not change anything.*

1.7 How to use promise theory

To apply promise theory to a new problem, one starts like this:

1. *Identify the key agents (devices and personal roles).*

   Choose an agent to be any part of your system that can change or act independently. To get this part of the modelling right, one must be careful not to confuse intentions with actions or messages[2].

   To be independent, an agent only needs to think differently or have a different perspective, access to different information, etc. This is about the separation of concerns. If we want agents that reason differently to work together, they need to promise to behave in a mutually beneficial way. In documenting this cooperation, one learns about the structure of intent in a system.

2. *Establish which agents make which promises*

   The goal of cooperation is to ensure that agents make all the promises necessary so that some imaginary on-looker (the 'god's eye view'), with access to all the information, would be able to say that an entire cooperative operation could be seen as if it were a single entity making a single promise, i.e. one aims to establish whether the sum of the parts leads to a promise of collective behaviour.

   Instead of thinking in terms of 'requirements', we transform into a mirror image of autonomous promises.

   If agents are not already motivated to work together somehow, then we want to know how we coax the agents to make and keep promises that fit into the larger plan. How this works depends on what kinds of agents they are. If they are human, economic incentives are usually the answer. If the agents are programmable, then they need to be programmed to try to keep appropriate promises. Either way, we call this voluntary cooperation.

3. *Allow for the uncertainty.*

   Promises might or might not be kept, for a variety of reasons beyond the control of any agent, so plan for that. Even a machine can break down and fail to keep a promise, so we need to model this. Each promise will have a probability associated with it, based on our trust or belief in its future behaviour[3].

4. *Eliminate conflicts of intent.*

   If all agents shared the same intentions, there would be no need for promises. Since the initial state of a system has unknown intentions, it means we have to set up things like 'agreements', where agents promise to behave in an aligned fashion. This is sometimes called orchestration of relative intent.

1.8 Rules of thumb summarizing promise oriented design

1.8.1 Thumb-rules for agencies

1. The bottom-up rule: dynamical composability

Put all promisable capabilities and properties into the smallest agents that can represent them, and inherit the promises upwards by aggregation.

e.g. in a choir, the promise of song should be associated with each member of the choir, and inherited upwards as an implicit promise of the aggregate, not vice versa.

2. The atomicity rule: semantic composability

   Represent every property, which can be determined independently, as a separate agent or component, and add promises to explain the cooperation with other components if necessary.

3. Originate or cache all required information locally: avoid dependence

   Plan for any information needed by an agent to be promisable by the agent itself, (either be or use a trusted source). Prefer not to rely on a remote agent, as that would introduce dependences and latencies that decrease the certainty of the promise being kept.

   Caching of dependent service lookups is thus a strategy for resilience and speed.

1.8.2 Thumb-rules for the promises

1. Map out the menu of states that an agent is capable of promising, and the subsets that would normally be used. This gives you a promise language vocabulary.
2. Break use-cases down into promises made up from this language.
3. Avoid conditional promises (they are fragile by dependency)
   (a) Conditionals increase complexity: a non-trivial conditional promise is cannot be realized without the help of additional agents. Conditional promises lead to dependencies (pre-conditions), and dependencies increase the number of ways a promise can fail.
   (b) Since any promise might not be kept, the probability of a failure is proportional to the number of promises. This scales approximately like N^2 where N is the number of agents.
   (c) Avoid serial (intermediate) agent chains, as these represent hard dependencies that increase both the number of agents and the number of promises.

   The probability of a chain of agents not keeping its promise is like the sum of probabilities for each agent in the chain (i.e. increases with the length of the chain) $\simeq Np \to 1$.

   Examples of serial dependency include independent load balancers, firewalls, delivery agents, etc.
   (d) Avoid combining depending on functions of one or more other (remote) agents. The probability of *keeping a promise* is proportional to the product of the probabilities of each agent, which grows smaller as a power law $\prod p_i \simeq p^N \to 0$.

   Examples of multiple dependence include dual key (two factor) authentication, microservice compositions, web pages built from multiple sources, such as database, advertising service, authentication scans etc.
4. Assume that common information known to different agents is inconsistent, i.e. that each agent has its own version which may or may not agree with one another. Do not rely on the *consistency* of information from multiple agents.
5. Keep promises separately for each timescale: match timescales for equilibrium

   The state of a promise (whether it is kept or not) has to be checked often enough to match the likely reasons for its failure. To maintain equilibrium, every property promised has its own timescale, and separate if the Δt is the MTBF, then the MTTR should be of the same order of magnitude Δt. By Nyquist's theorem, we should sample and check for repair at least twice as fast as the fastest change that can cause a promise to not be kept.
6. Fault modes: every promise is a potential fault

   Promises can be broken actively (by introducing conflicting promises) and passively (by not repairing the state of a promise that is not being kept).

1.8.3 Scaling promises and services

1. Scaling usage at constant reliability:

   For every use (-) promise made by a client agent, assume a parallel provider agent (+) will be required for scale invariance.

   Scale promises by adding indistinguishable, interchangeable agents, i.e. agents that make the same promises in parallel.

2. Scaling information to reduce complexity:

   Use promise roles to hide detail, and refer to groups of equivalent agents as super-agents.

   The consumer of an array of agents can treat all the parallel agents as a single effective system provided there is an index or directory that promises to reveal connect the consumer of a promise to the provider of a promise.

1.8.4 Knowledge management

1. Comprehensibility: clear description

   Make sure the description of a promise outcome is easy to comprehend for humans as well as the agencies that formally keep the promise. The information about which promises are kept is for *human consumption*.

2. Adaptability: designing in context

   Try to separate contexts into different promises, conditional on the context. To avoid conditional dependency, make sure that every promise has a default context.

3. Flexibility: usability semantics

   Flexibility comes from the ability to use promises in different contexts. This is simpler if different promises are made by separate agencies, so the composability is maximized.

   e.g. This is like microservice design in software engineering.

4. Traceability: understanding cause-effect

   By following the basic rules above for agency and promises, every detail in a system of promises will be mapped out clearly. Clear promises between independent components document all the causal channels. This maximizes the understanding of fault modes, and how they can be mitigated.

5. Handling uncertainty: contingency and redundancy

   Never assume that a promise will always be kept. Take all uncertainty about the ability to keep promises into account when designing a system.

   If you are designing a component, make sure it is either intrinsically reliable, by internal redundancy, or can be used in parallel to achieve fault tolerance. Even when you've done this, it can fail, so design a contigency..

   Fault theorem: low level reliability is always as good as or better than high level redundancy (see figure 1.2).

1.9 Timescales

A promise model of managing a device, which is supposed to keep a number of promises based on promises. There is a slow governance process, with promised compliance, and a fast workload process, with service level promises.

The change model is assumed to wor

- Rapid fluctuating changes, e.g. jobs starting and stopping, fluctuations on top of a stable processing system.
- A steady state equilibrium for handling policy constraints, punctuated by occasional changes.
- Intentional interventions, or policy changes driving a trend of slow but steady improvement

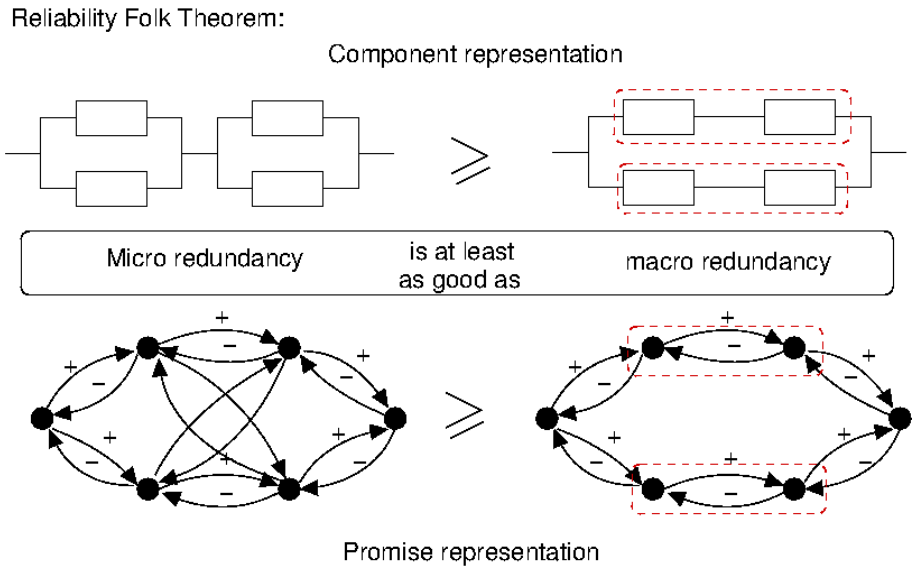

Figure 1.2: Component and promise diagrams illustrating the well-known result of reliability theory, that low level redundancy is always at least as good as high level.

- Cycle of continuous improvement of policy: a value gradient climbing by decisions.

We can assume that there is some agency "agent" which is responsible for keeping the promises. It is easier and more reliable to keep these promises if this agent is part of the device itself, but it could operate by remote control in a more fragile model.

The interpretation of this diagram depends on the timescale of the loops that maintain equilibrium. For the slow queue (1), unintended change enters the systems forming a queue to be balanced by the agent. For the faster job queue (2), the unplanned change is the arrival of a job. This has to be balanced by the scheduler.

By Nyquist's theorem, the maintainence process has to be at least twice as fast as the fastest continuous change expected, in order to keep pace.

- The promises that govern the fast loop are basically fixed by the design of the service.
- The promises that givern the slow loop are subject to tuning, due to on-going adaptation.

In general, the distance to a controller should be minimized to optimize the performance. If feedback is slow, allocation time will only proceed as the rate of the slow feedback, instead of the rate of the arrival process.

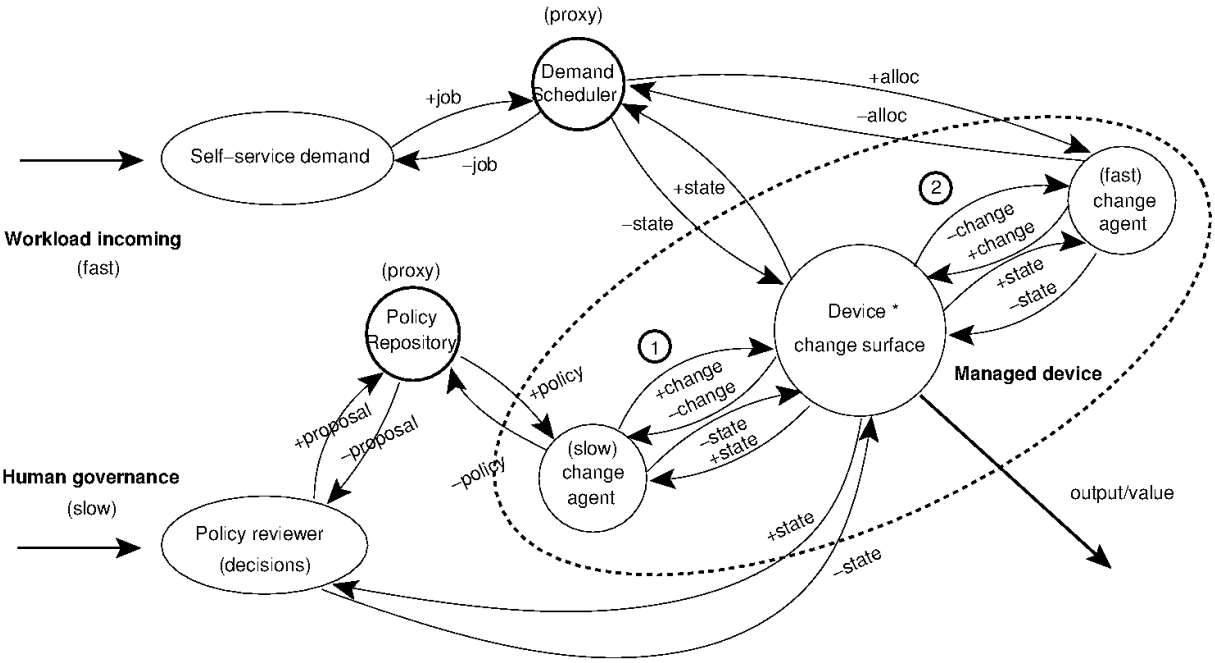

Figure 1.3: A promise diagram showing the promise relationships. Each arrow is a promise. Note the balance between promises to give (+) and promises to accept (-). The endless cycle of continuous value creation for policy, and the one-time cycle of resource allocation for ephemeral tasks.

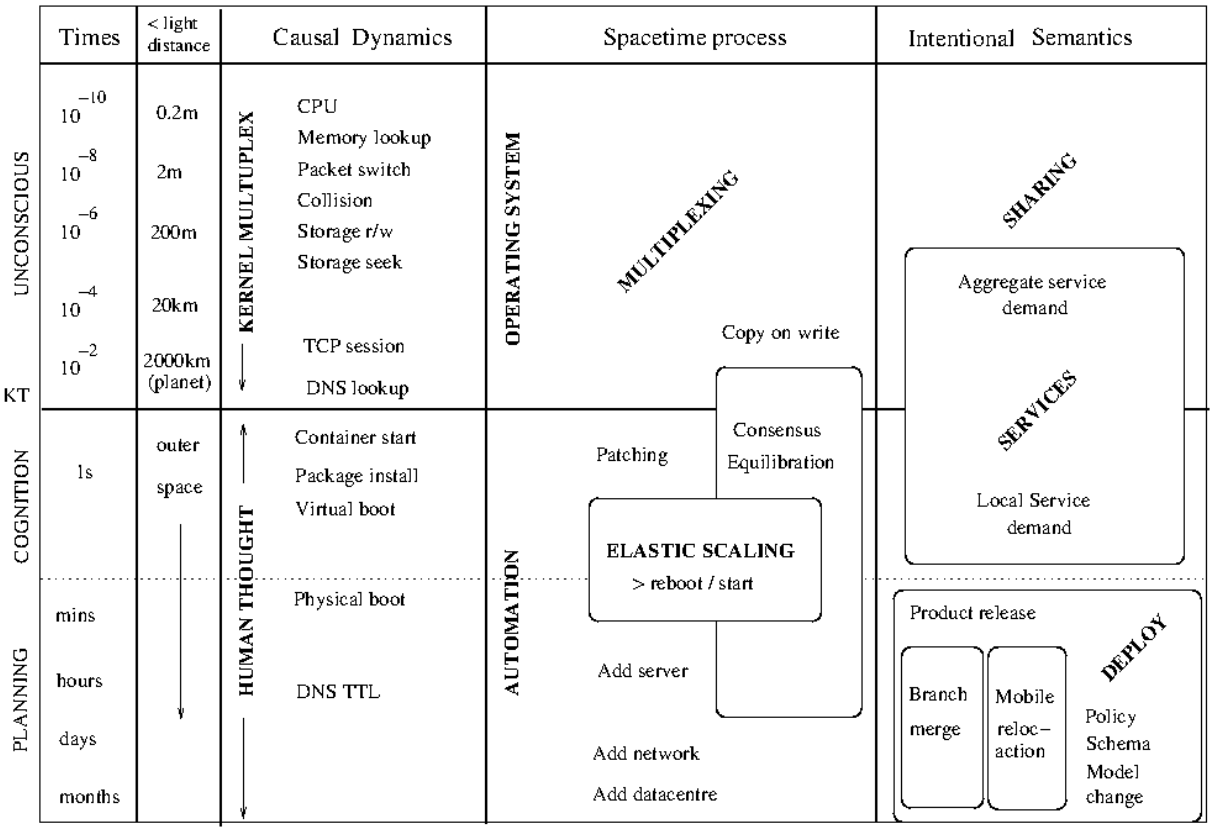

Figure 1.4: Common timescales in infrastructure.

Chapter 2

Promise syntax representations

2.1 Promise notations

We need a way to write promises that is sufficiently formal to be rigorous. It is handy to use some different formats, or alternative syntaxes.

- Graphlcal: promises are relationships and have a natural graph structure. A graphical view often illustrates channels of causation.
- Algebraic: Pictures are combersome to label, so a graphlike algebraic notation is useful as a compact way of describing promises.
- As a data structure: In information technology, we like to reduce descriptions to the familiar data structures of programming languages.

Whichever of these forms we choose, the results should be the same.

2.2 Promises involving groups, anonymous and wildcard agents

We can define notation for a promise from one set of nodes to another set of nodes using set curly-braces:

$$\{A_i\} \overset{b}{\rightarrow} \{A_j\} \tag{2.1}$$

i.e. the set of agents $\{A_i\}$ promises b to another set $\{A_j\}$ of agents.

We can now suppose what happens when referring to third-party agents in the body of a promise. In some circumstances, this can apparently lead to contradictions. As a general rule, it seems to be in the interest of clarity to avoid this where possible.

The general promise notation above is designed to avoid the problem of referring to other agents in the body of a promise, but to make some promises we need to talk about third parties. Let us denote:

- Unspecified agents "$A_?$"
- Anonymous agents "$A_{\rm anon}$".

Ensembles of these can be denoted with braces, e.g. $\{U\}$ would be an ensemble of unspecified agents. We can write

$$A_{\rm anon}(\{A\}_1) \overset{b}{\rightarrow} A_{\rm anon}(\{A\}_2) \tag{2.2}$$

to mean one anonymous agent from the ensemble $\{A\}_1$ makes a promise to another anonymous agent from the ensemble $\{A\}_2$.

2.3 CFEngine representation

The CFEngine language has two goals: to represent promises and to model resource patterns.

Its basic syntax represents fixed the promises that device resources can make, and the structure and grammar enable the representation of compression through pattern matching. Without the latter, the configuration for a network of collaborating devices would be unwieldy.

The CFEngine syntax is a flat key-value language, which highlights promiser and promisees, and has a fixed type set. The promiser and promisees are named in whatever convention is used by the system, e.g. filesystem names, URIs, process names, storage devices, etc. Aliases can be improvised using the general capacity to make variables and manipulate data.

- Promises that manipulate arbitrary data (for pattern matching, etc).
- Promises about the state of the system (providing context for conditional/relative promises)
- Promises about resource state.
- Promises about CFEngine's modus operandi.

There is no type definition in CFEngine, only instance definition. body - re-usable template (parameterizable), like grouping but for instances

```
promiser -> { promisees ... },

      attribute1 => "value1",
      attribute2 => "value2",
      attributeN => "valueN";
```

To avoid endless recursion (such as in XML), the CFEngine language avoids explicitly indentifying several compound structures, such as the promise body. This has advantages and disadvantages. It makes the syntax clear and simple, but it does violence to the intuitions of modern programmers. Thus, one could have added more explicit containerization of the syntax:

```
promiser -> { promisees ... },

  body
   {
      // Constraints
      attribute1 => "value1",
      attribute2 => "value2",
      attributeN => "valueN";
   }
```

CFEngine also avoids defining input types too strictly, preferring a runtime typing model for the most part. Thus, we have string, int and real as late-binding categories for approximate pre-validation. The intricacies of variables expansion in the pattern evaluation makes runtime typing difficult to achieve[1].

Some constraints are related to the functioning of the OM system (these are called "controls"), others relate to the promises that the OM system try to keep.

2.3.1 Common promise types (all modules)

vars defaults classes (runtime conditions) reports / exceptions / logs

[1] YANG, by contrast, is a pure data language - without sophisticated pattern generation capabilities.

2.3.2 Module specific promise types (examples)

Model resource types, and 'universal' attributes, rather than device capabilities. Some extensions for popular variations are supported, but mostly standards based, e.g. POSIX, or IETF.

- **files**
- **processes**
- **interfaces**
- **jobs/executions**
- **storage**

This is in contrast to CIM and SNMP which model by device type.

2.3.3 Common promise attributes - internal promises

2.3.4 Common promise attributes - user-defined promises

Actions/remediation

Examples of triplets that describe choices for promises that are hard-coded into the CFEngine software

```
attribute "action_policy",
values "fix,warn,nop",
description "Whether to repair or report about non-kept promises"

attribute "ifelapsed",
value CF_VALRANGE,
description "Number of minutes before next allowed assessment
             of promise. Default value: control body value"

attrubute "expireafter",
value CF_VALRANGE,
description "Number of minutes before a repair action is interrupted
                  and retried. Default value: control body value"

attribute "log_string",
value CF_ANYSTRING,
description"A message to be written to the log when a promise
               verification leads to a repair"
```

Class/context evaluation

Agents can make decisions based on their understanding of the current state of the world. Promises are often made relative to such a context. This is a form of conditionality, which is more stable than decision logic.

2.3.5 Module specific promise attributes

Chapter 3

BGP as promises

BGP is a promise based systems. Its autonomous agencies are routers and "autonomous systems" (AS), which make promises to one another.

3.1 The agents

The agencies of BGP, from a promise theory perspective are:

1. The Autonomous Systems ('AS'), numbered from 1..65535
2. The routing device hosts (characterized by router ID or 'loopback address')
3. The interfaces of the routing hosts (characterized by IP addresses)

The first two are just as one would expect from routing protocols, The latter (interfaces) have to be considered separate agencies, since they can make promises directly, as independently addressable entitites.

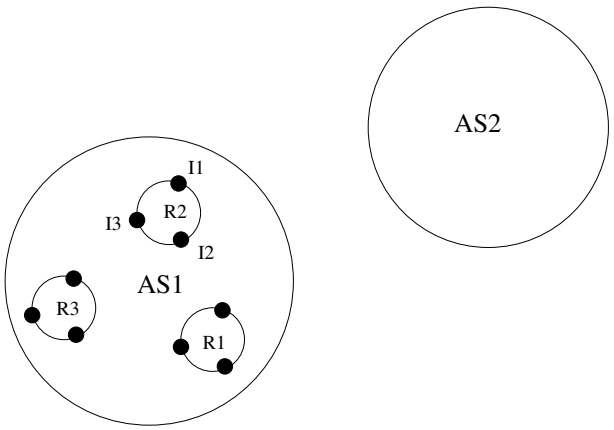

Figure 3.1: BGP agencies: AS, R(outers) and I(nterfaces) makes standalone promises, and also promises between one another in order to work as a system. The AS is a formal agent, formed by cooperation. It has no independent agency to keep promises without relying on its resident router members.

The promises

1. The Autonomous Systems ('AS')
   - Must promise a number from 1..65535
   - Can promises routing information (advertisements)
   - Can promise to relay traffic by exposing path information

2. The routing device hosts (characterized by router ID or 'loopback address')
    - Must promise to run the BGP service
    - Promises to accept/use BGP service (access control)
    - Promises to keep BGP policy promises, such as peering with neighbours etc
3. The interfaces of the routing hosts (characterized by IP addresses)
    - Can promise an IP address
    - Can promises a peer connection over unnumbered link

3.2 IoS configuration language

Focuses on the changes of state instead of on the state of the router itself. On the other hand, it assumes to maintain the state of the CLI command hierarchy

```
 configure terminal
    router bgp 65000
    router-id 0.0.0.1

configure terminal
   router bgp 65000
   neighbor 10.0.0.2 remote-as 65002

 configure terminal
    router bgp 65000
    neighbor swp1 interface
    neighbor swp1 remote-as 65100

 configure terminal
    router bgp 65000
    neighbor swp2 interface
    neighbor swp2 remote-as 65102

configure terminal
  router bgp 65000
  neighbor 2001:db8::ff33 remote.as 65000

configure terminal
  router bgp 65000
  neighbor 2001:db8::ff33 route-reflector-client

configure terminal
   router bgp 65000
   redistribute kernel

configure terminal
   router bgp 65000
   address-family ipv4 unicast
   neighbor 10.0.0.2 activate

configure terminal
  router bgp 65000
   address-family ipv6
   network 2001:0DB8:AAAA:1::/64
```

These are all promises made by the router on which the commands are entered. But as promises, we can collect the configuration desired states of all routers into a single place.

3.3 Configuration summary shows promises more clearly

Because the IoS CLI is designed (implicitly) by a state machine model, when the final state is dumped, it takes on the appearance of a set of promises:

```
!
router bgp 65000
 bgp router-id 0.0.0.0
 bgp graceful-restart
 network 10.0.0.0/24
 network 20.0.0.0/24
 neighbor 10.0.0.2 remote-as 65002
 neighbor 10.0.0.2 advertisement-interval 0
 neighbor 10.0.0.2 next-hop-self
 neighbor 2001:db8::ff33 remote-as 65000
 neighbor 2001:db8::ff33 route-reflector-client
 neighbor 2001:db8::ffff remote-as 65001
 neighbor 2001:db8::ffff ebgp-multihop 22
 neighbor 2001:db8::ffff advertisement-interval 1
!
 address-family ipv6
 network 2001:db8:0:2::/64
 network 2001:db8:aaaa:1::/64
 network 2001:db8:aaaa:2::/64
 neighbor 10.0.0.2 activate
 neighbor 2001:db8::ffff activate
 neighbor 2001:db8::ffff soft-reconfiguration inbound
 exit-address-family
!
line vty
!
end
```

This configuration can be reloaded into a single routing device, as a policy document. This gives clues about how to formulate a proper policy language that models an entire distributed system.

3.4 Two peer session example

The promises of type 1 and 2, associated with the bgp service running on the routing host, can be summarized with only a small amount of configuration data. Instead of having a separate configuration for each router, these may be combined into a single source contextual model.

```
body routing_services control
{
router1:: # context by name

 bgp_local_as => "65001";
 bgp_router_id => "1.1.1.1";
 bgp_ipv4_networks => { "10.10.10.0/24", "10.20.30.0/24"  };

router2:: # context by name
```

```
 bgp_local_as => "65002";
 bgp_router_id => "2.2.2.2";
 bgp_ipv4_networks => { "20.20.20.0/24"  };

cumulus:: # context by device operating system

 routing_service_log_file => "/var/run/log/quagga/bgpd.log";
 bgp_graceful_restart => "true";

any:: # common for all

 bgp_log_neighbor_changes => "true";
 bgp_redistribute => { "kernel", "static", "connected" };

}
```

Interface specific promises are related to sessions between devices.

```
bundle agent main
{
interfaces:

 router1::

   # Associate the session with the p2p connection

   "port47"
       link_services => ebgp_session("192.168.47.2", "65002");

 router2::

   "port23"
       link_services => ebgp_session("192.168.47.1", "65001");

}
```

Note that, since this is peer to peer or point to point, there is no real need to use the IP addresses of the sessions endpoints. Unnumbered interfaces can be used to eliminate one more piece of the promise data. Instead of passing the IP address as a parameter to the template/pattern, we can simply use the self-reference to the promiser `$(this.promiser)`.

Then in the configuration of the service agency, which is common to all interfaces (rather than of the interfaces individually), we can add router/host specific promises.

```
body link_services ebgp_session(directip, remoteas)
{
bgp_session_neighbor => "$(directip)";
# or bgp_session_neighbor => "$(this.promiser)"; for unnumbered

bgp_peer_as => "$(remoteas)";
bgp_ttl_security => "1";
bgp_advertisement_interval => "0";
bgp_external_soft_reconfiguration_inbound => "true";
bgp_advertise_families => { "ipv4_unicast" };
}
```

3.5 Example Cumulus/CFEngine policy to configure BGP fabric

```
################################################
#
# eBGP wan peering example - 2 AS talking to each other
#
#        R2
#      /
#   R1

################################################

bundle agent main
{
interfaces:

 cumulus_switch1::

   "swp47"

       link_services => ebgp_session("192.168.47.100", "65002");

 ipv4_192_168_47_100::

   "eth0"

       link_services => ebgp_session("192.168.47.1", "65001");
}

################# NODE SERVICE SETTINGS ######################

body routing_services control
{
cumulus_switch1::

 bgp_local_as => "65001";
 bgp_router_id => "1.1.1.1";
 bgp_ipv4_networks => { "10.10.10.0/24", "10.20.30.0/24"  };

ipv4_192_168_47_100::

 bgp_local_as => "65002";
 bgp_router_id => "2.2.2.2";
 bgp_ipv4_networks => { "20.20.20.0/24"  };

any:: # common for all

 routing_service_log_file => "/var/run/log/quagga/bgpd.log";
 bgp_log_neighbor_changes => "true";
 bgp_redistribute => { "kernel", "static", "connected" };
}
```

3.6 2 spine by 5 leaf Clos patter

```
bundle agent main
{
vars:

 "spine"  slist => expandrange("swp[1-5]", "1"); # point to 5 leafsw
 "leaves" slist => expandrange("swp[1-2]", "1"); # point to 2 spinesw

 "net_adverts[leaf1]" slist => { "10.10.10.1/24", "10.10.20.1/24" };
 "net_adverts[leaf2]" slist => { "10.10.30.1/24", "10.10.40.1/24",
                                 "2001:0DB9:0:f101::1/64" };
 "net_adverts[leaf3]" slist => { "192.168.1.0/24" };
 "net_adverts[leaf4]" slist => { "192.168.1.0/24" };
 "net_adverts[leaf5]" slist => { "192.168.1.0/24" };

 "router_id[spine1]" string => "2.0.0.1";
 "router_id[spine2]" string => "2.0.0.2";
 "router_id[leaf1]" string => "1.0.0.1";
 "router_id[leaf2]" string => "1.0.0.2";
 "router_id[leaf3]" string => "1.0.0.3";
 "router_id[leaf4]" string => "1.0.0.4";
 "router_id[leaf5]" string => "1.0.0.5";

 ...
```

Using the pattern in these data, it is a simple matter to define iterators to generate the pattern in context, without having to expand the entire configuration up front. This kind of late/lazy evalutation leads to huge savings in management overhead.

```
...

interfaces:

 spine::

   "$(spine)"
       link_services =>  ibgp_reflector("server");

 ToR::

   "$(leaves)"
       link_services =>  ibgp_reflector("client");

}
```

As before, there is a template for the interface promise body:

```
body link_services ibgp_reflector(role)
{
bgp_session_neighbor => "$(this.promiser)";
bgp_peer_as => "65000";
bgp_route_reflector => "$(role)";
bgp_ttl_security => "1";
bgp_advertisement_interval => "0";
bgp_internal_next_hop_self => "true";
bgp_advertise_families => {"ipv4_unicast", "ipv6_unicast"};
```

```
bgp_maximum_paths => "64";
}
```

And the router-specific promises are made on the assumption of standard internal promises, defined according to the BGP standard and the IoS de-facto standard. In iBGP, all nodes have the same share the same AS number.

```
body routing_services control
{
inside_my_AS::
 bgp_local_as => "65000";
 bgp_router_id => "$(clos.router_id[$(sys.uqhost)])";
 routing_service_log_file => "/var/run/log/quagga/bgpd.log";
 bgp_ipv4_networks => { @(clos.net_adverts[$(sys.uqhost)]) };
 bgp_redistribute => { "kernel", "static", "connected", "ospf" };
 bgp_graceful_restart => "true";
}
```

3.7 Advantage of an explicit promise viewpoint

BGP is built on promises, but the command language to configure it is not. This lead to an overcomplication, and an inability to deal with more than one box at at time.

By combining into contextual promise model, all of the configurations for an arbitrarily large number of routers/devices, can be combined with configurations for servers and other devices, leading to a single map of infrastructure.

Chapter 4

CFEngine examples on servers

4.1 Database structure/content promises

```
bundle agent databases
{
  databases:

      "cfengine.db/brand_new"

      # These are the defaults
      # database_type => "sql",
      # database_operation => "create",

      database_columns => {
                           "mighty varchar(50)",
                           "killer varchar(80)",
                           "batman varchar(20)",
                           "rating integer",
      },
      database_rows => {
                         "mighty='mouse', killer='cheeses', batman => 'yes', rating=>'5' ",
                         "mighty='gort', killer=>'raygun', batman => 'yes', rating=>'10' ",
                         "mighty='fred', killer='queen', batman => 'no', rating=>'10' ",
      },

      database_server => sqlite;
}

#
# Templates allow for context dependent paramterizations
#

body database_server sqlite
{
      db_server_type => "sqlite";
      db_embedded_directory_path => "/tmp";
}

body database_server local_mysql(username, password)
{
      db_server_owner => "$(username)";
      db_server_password => "$(password)";
```

```
      db_server_host => "localhost";
      db_server_type => "mysql";
      db_server_connection_db => "mysql";
}


body database_server local_postgresql(username, password)
{
      db_server_owner => "$(username)";
      db_server_password => "$(password)";
      db_server_host => "localhost";
      db_server_type => "postgres";
      db_server_connection_db => "postgres";
}
```

4.2 Container deployment

```
bundle agent my
{
vars:

  "containers" slist => { "parent_container_1",
                          "parent_container_2",
                          "parent_container_3" };

guest_environments:

   "$(my.containers)" -> { "application_group" }

       guest_details => ubuntu_stem_cell,
       guest_state   => "create";

  # Reap garbage

   "old_container.*"

       guest_details => ubuntu_stem_cell,
       guest_state   => "delete";
}

body guest_details ubuntu_stem_cell
{
guest_type => "docker";
guest_image_name => "cf-stem-cell";
}
```

4.3 Example Monitoring of counts and streams on a Linux platform

```
#########################################################

bundle monitor watch
{
```

```
  measurements:

      # Test 1 - extract string matching

      "/home/mark/tmp/testmeasure"

      handle => "blonk_watch",
      stream_type => "file",
      data_type => "string",
      history_type => "weekly",
      units => "blonks",
      match_value => find_blonks,
      action => sample_min("10");

      # Test 2 - follow a special process over time
      # using cfengine's process cache to avoid resampling

      "/var/cfengine/state/cf_rootprocs"

      handle => "monitor_self_watch",
      stream_type => "file",
      data_type => "int",
      history_type => "static",
      units => "kB",
      match_value => proc_value
        (
        ".*cf-monitord.*",
        "root\s+[0-9.]+\s+[0-9.]+\s+[0-9.]+\s+[0-9.]+\s+([0-9]+).*"
        );

      # Test 3, discover disk device information

      "/bin/df"

      handle => "free_disk_watch",
      stream_type => "pipe",
      data_type => "slist",
      history_type => "static",
      units => "device",
      match_value => file_system;
      # Update this as often as possible

      # Test 4

      "/tmp/file"

      handle => "line_counter",
      stream_type => "file",
      data_type => "counter",
      match_value => scanlines("MYLINE.*"),
      history_type => "log";

}

#########################################################

body match_value scanlines(x)
{
```

```
      select_line_matching => "^$(x)$";
}

############################################################

body action sample_min(x)
{
      ifelapsed => "$(x)";
      expireafter => "$(x)";
}

############################################################

body match_value find_blonks
{
      select_line_number => "2";
      extraction_regex => "Blonk blonk ([blonk]+).*";
}

############################################################

body match_value free_memory # not willy!
{
      select_line_matching => "MemFree:.*";
      extraction_regex => "MemFree:\s+([0-9]+).*";
}

############################################################

body match_value proc_value(x,y)
{
      select_line_matching => "$(x)";
      extraction_regex => "$(y)";
}

############################################################

body match_value file_system
{
      select_line_matching => "/.*";
      extraction_regex => "(.*)";
}
```

Chapter 5

GBP - Group based policy

Group based policy is a policy framework for implementing a promise-like policy on networked applications. It is used by Openstack, and Open Daylight, for instance, and is related to the Cisco ACI policy model.

It is an applucation centric viewpoint, rather than a hardware box viewpoint. This means the primaru focus is on what the applications promise, with dependence on what hardware can deliver. Applications run as processes, possibly wrapped in containers like Docker, virtual machines, etc, and run on top of hardware.

GBP is modelled naturally by promises, i.e. without unnecessary overriding of locality, etc, because:

- In GBP the agencies are autonomous, i.e. they make local decisions, thus policy can be decomposed into autonomous components.
-

5.1 Nomenclature

Referring to the figures, we can compare some of the terminology for GBP and promises.

Description	GBP	Promises/CFEngine
Aggregate policy element	Contract	Promise bundle
Affected item	Subject	Promiser (agent)
Stakeholder in outcome	-	Promisees (agents)
Policy actuation	Action	Promise repair (must converge)
Parameters and semantics	Clause / Directive	Promise body
(desired outcome)	(set of actions)	(key-value end-states)
Kind of item	Clause attribute?	Resource type
Attribute declaration	Subject rule	Promise body constraint
Applicability circumstances	Classifier	Class/context probe
Give	Provide (not used)	(+) Promise to provide
Take	Consume	(-) Promise to use

5.2 GBP as promises

Because GBP is tied to IT, especially networking, many of the constructs are ‘hardwired’ to represent interacting appliucations.

Let’s compare it to a promise model. The agencies in the promise model are processes that execute parts of an application service. In the Open Daylight version, the policy is only about networking and NFV, which means that half of the promises (that provide services) are not represented.

- What are the agencies (not just the hardware boxes, but the software entities) that keep promises?
- What dependencies are engineered into the components?

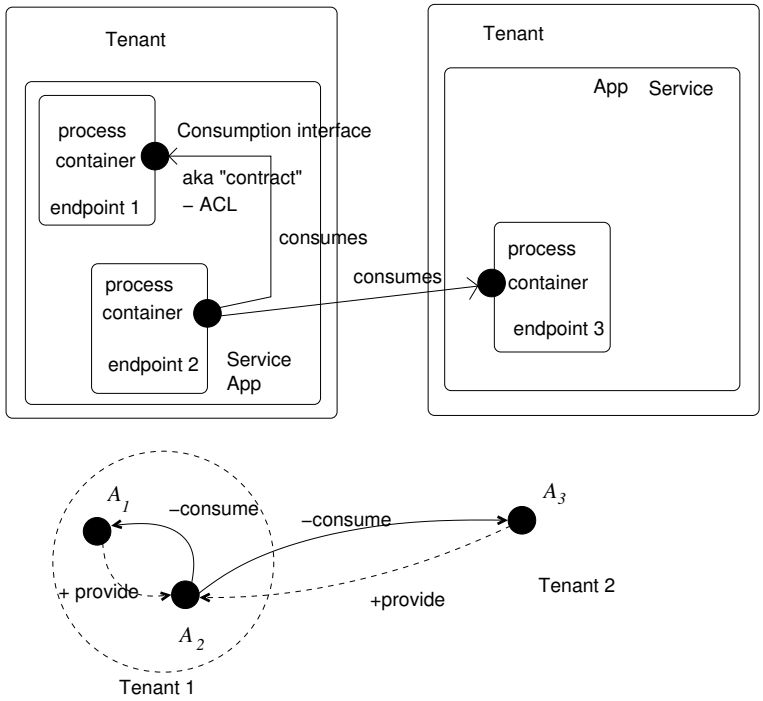

Figure 5.1: Comparing GBP with promise agency. In the BGP structure (above), each box represents a data structure in a hierarchy. Promise theory (below) focuses on the atoms that have actual agency, i.e. are capable of managing state. The aggregations and abstractions above are mainly window dressing and so may be ignored.

5.3 Promise convergence, versus general actions

A sequence of general responses to conditions can lead to any kind of outcome. There does not have to be any constraint on the state of the system after an abitrary number of actions matching certain conditions. To avoid the system becoming unstable, we therefore require promises to be represented by convergence outcomes.

Convergence of promises means that the number of allowed states after a promise is kept must be less than or equal to the number of allowed states before, so that the scope of behaviour is narrowing into an allowed target set.

$$\text{States after promise} \leq \text{States before promise} \tag{5.1}$$

This is a constraint on the kinds of promises a system should be allowed to make, for stability. This does not seem to be represented in GBP.

5.4 Policy constraints and compatibilty

Promises cannot have distributed conflicts. GBP should not be able to have distributed conflicts either, since it is *local*, like a promise model.

According to the information wiki, the policy expressed by GBP is a combination of three inputs:

- The applications desires
- The network's capabilities
- The infrastructure policy

though it does not explain how this works, or indeed what 'infrastructure' means. Let's look at a promise model. Let A_i by an application component, running on infrastructure host I_i, and connected by network N_{ij}

An application component A_1 can promise some application service a to A_2, conditionally on its dependencies I, N, which promise i for infrastructure resources and n for network resources, respectively (see fig. 5.4). By the

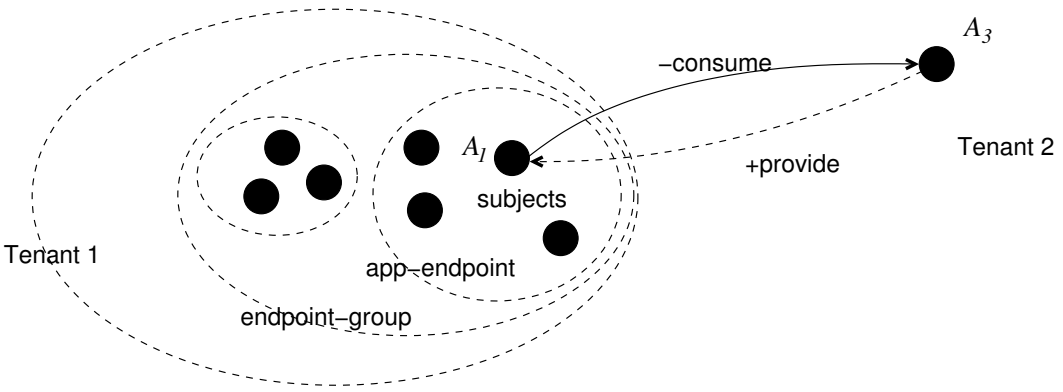

Figure 5.2: In a promise model, the proper agency builds from the bottom up. The aggregate structures shown by dotted lines are mainly logical boundaries, and generally bring no additional properties. The boundaries can, themselves, be represented as promises on the subject agents.

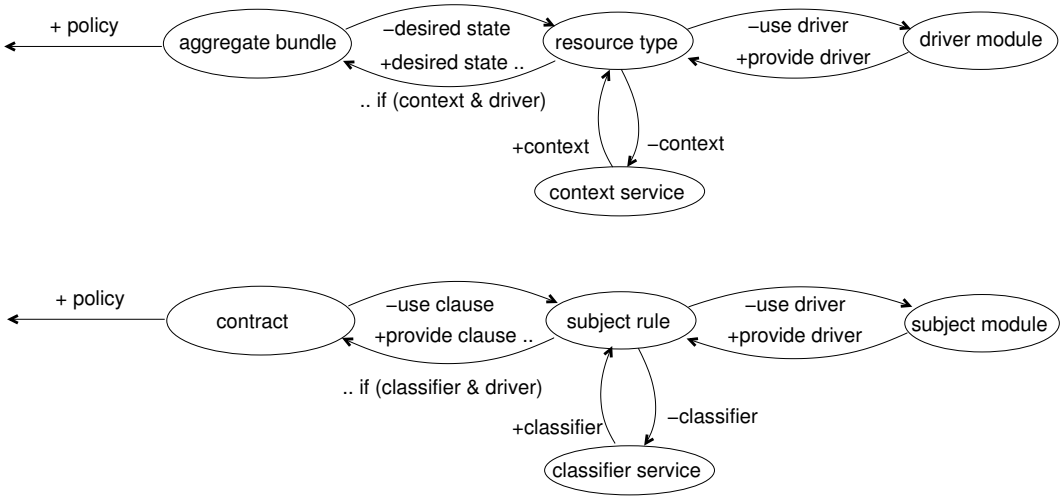

Figure 5.3: Comparing CFEngine promise terminology and policy structure with Group Based Policy nomenclature

rules of conditional promising, there must be promises by I, N accepted by A_1 and A_2

$$A_1 \xrightarrow{+a_1|I,N} A_2 \tag{5.2}$$
$$I \xrightarrow{+i} A_1 \tag{5.3}$$
$$N \xrightarrow{+n_{12}} A_1 \tag{5.4}$$
$$A_1 \xrightarrow{-i_1} I \tag{5.5}$$
$$A_1 \xrightarrow{-n_1} N \tag{5.6}$$

So the service level that A_1 can promise is

$$\max a_1 = a_1 \cap i_1 \cap n_1 \cap i \cap n_{12} \tag{5.7}$$

Then. receiving by A_2, with promise

$$A_2 \xrightarrow{-a_2|I',N} A_1 \tag{5.8}$$
$$I' \xrightarrow{+i'} A_2 \tag{5.9}$$
$$N \xrightarrow{+n_{12}} A_2 \tag{5.10}$$

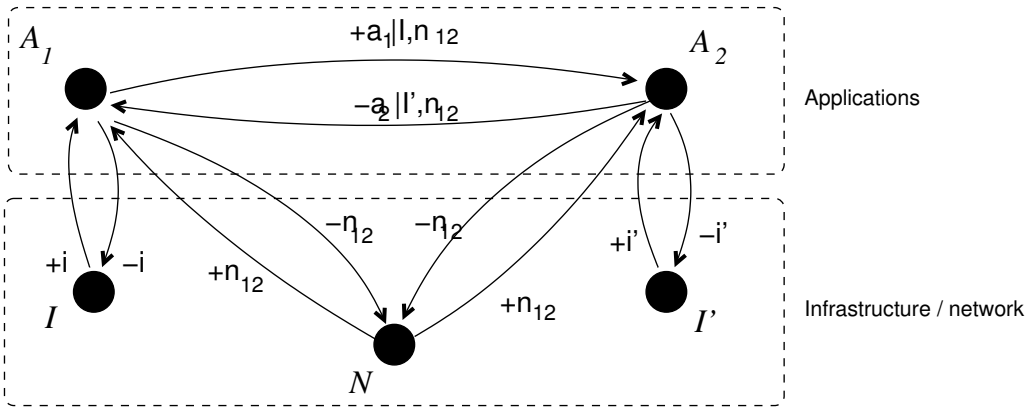

Figure 5.4: Promises between applications that rely on infrastructure promises. This follows the classic intermediate agent pattern.

Hence the maximum service level receivable by A_2 from A_1 is:

$$\max a_2 = a_1 \cap i_1 \cap n_1 \cap i \cap n_{12} \cap i' \cap a_2. \tag{5.11}$$

5.5 What might GBP look like in CFEngine style promise language?

5.5.1 Promise: packet/flow access

We want to be able to say things like: "Allow TCP/80 from certain addresses, with maximum flow rate 20MB/s, etc".

i.e. I promise to allow packets on TCP/80 from certain addresses up to a maximum average rate, and/or maximum amount of data.

- What agency is going to keep the promise? (A gateway of some kind)
- How can the promise be parameterized, in the general case? (Promise body)

```
gateway:

  "veth0"   -> stakeholders,

     # anything looking likf f(..) is a parameter bundle

      select_packets => filter("TCP","80","outgoing"),
     allow_from => {  "example.org", "123.456.789.0/24" },
      deny_from => { "123.456.789.6" },
     limit_rate => mySLA("20","100000");
```

5.5.2 Promise: data gravity

We want to be able to say things like: "please make sure the containers you schedule for my job are as close as possible to their data source".

i.e. I promise that I am going to use significant amount of data from this source (so it is in your own interest to put me close to it).

- Virtual compute node

- How can the promise be parameterized, in the general case? (Promise body)

```
guest:

  "container"   -> stakeholders,

      close_to => locate_data("mydata");
```

5.5.3 Promise: network privacy

We want to be able to say things like: “please make sure any connections to be are over a secure line”.
i.e. I promise to accept connections, if and only if they satisfy these requirements.

- Application container’s virtual interface
- How can the promise be parameterized, in the general case? (Promise body)

```
gateway:

  "veth0"   -> stakeholders,

       # some criteria for even allowing a connection

      require_encryption => minimium_allowed("AES512","BC");
```

Chapter 6

YANG modelling language as a promise system

6.1 YANG representation

The YANG modelling language (written for NETCONF) is a reasonably generic representation of states and transitions which can be used to represent promises.

It is not unlike the CFEngine language, but more general in some ways, while being tied closely to hierarchical XML representation style.

A YANG model is not a promise model unless it satisfies the promise principles of *node autonomy* and voluntary cooperation.

Supports namespaces and modules (with prefixes)

```
module ietf-inet-types
{
namespace "urn:ietf:params:xml:ns:yang:ietf-inet-types";
prefix "inet";

organization "IETF NETMOD working group";
description "YANG data types";
revision 2010-09-24
   {
   reference "RFC 6021";
   description "Initial commit";
   }
}
```

Groups as aggregates - like classes in CFEngine
Configuration data models - these are prewritten modules
Data models in YANG

- IPFIX configuration model
- NETMOD interfaces core model
- NETMOD IP and routing core models
- NETCONF Netconf monitoring model
- NETCONF Netconf access control model

6.1.1 Statements

- **leaf**: Has one value, no children, one instance

```
leaf hostname
{
type string;
mandatory true;
config true;
description "Hostname for this system";
}
```

This maps to XML like

```
<hostname>my.example.com</hostname>
```

- **leaf-list**: Has one value, no children, multiple instances. This maps to XML like

```
<hostname>my1.example.com</hostname>
<hostname>my2.example.com</hostname>
<hostname>my3.example.com</hostname>
```

- **container**: Its value is to hold related children one instance

```
container system
   {
   container services
      {
      container ssh
         {
         presence "Enables ssh";
         // more leaf items ....
         }
      }
   }
```

This maps to XML like

```
<system>
  <services>
     <ssh>
       ...
     </ssh>
  </services>
</system>
```

- **must** Constraints nodes (leaves?) by an XPATH expression

```
container timeout
   {
   leaf access-timeout
      {
      description "Max time ...";
      units seconds;
      type uint32;
      }

   leaf retry-timer
      {
```

```
      description "Retry period";
      units seconds;
      type uint32;
      must "$this < ../access-timeout"
         {
         error-app-tag retry-timer-invalid;
         error-message "The retry timer must be"
                    + "less than the access timout";
         }
      }
   }
```

- **list** Key value structure, like a JSON tag.

```
list user
    {
    key name;   // name is the primary key
    leaf name
       {
       type string;
       }

    lead uid
       {
       type uint32;
       }
    }
```

- **augment-when** is like a class context expression in a promise body, enabling conditional body types.

```
augment system/login/user

    when "class = wheel";
    leaf shell

       type string;
```

- **grouping** is a re-usable node group, a pseudo typedef aggregate, without replacing by a new name

```
grouping target

   leaf address

      type inet:ip-address;
      description "target ip address";

   leaf port

      type inet:ip-port;
      description "target port";



// ...
```

```
list nameserver

    key "address port";
    uses target;
```

This maps to XML

```
<nameserver>
   <address>192.168.1.1</address>
   <port>80</port>
</nameserver>
```

6.2 Disadvantages of YANG

It is tied to concepts in XML, despite having translations to JSON and other formats

It is a container based model rather than a flat parallel model. Hierarchical ontology is encouraged.

Chapter 7

Application areas

7.1 Firewall

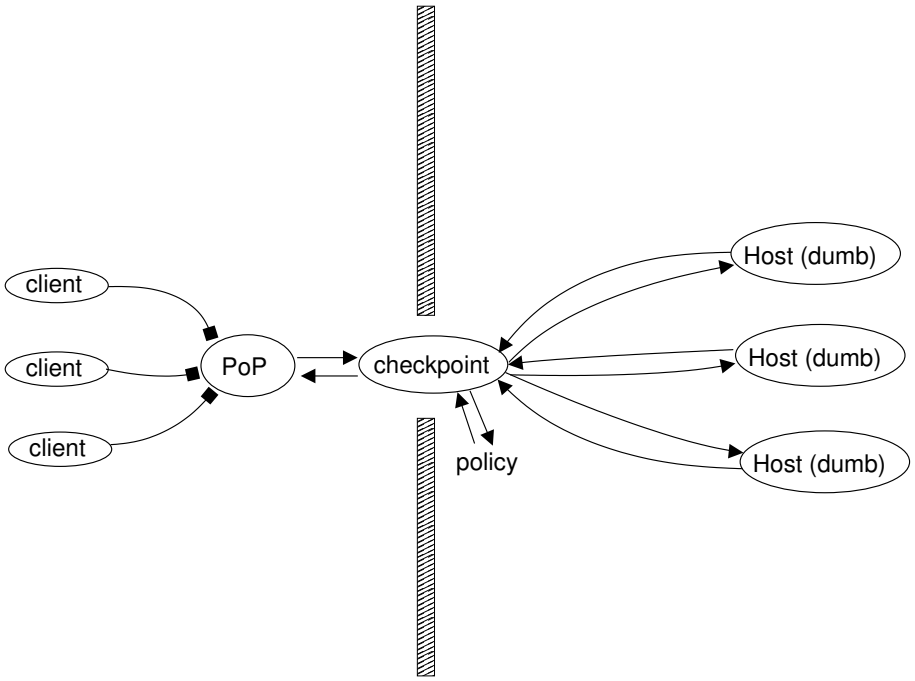

Figure 7.1: Firewall old.

7.2 Load balancer

7.3 FCAPS

The usual: faults, configuration, accounting, performance and security.

7.4 IPAM

DNS, DHCP, ARP, Ipv6,

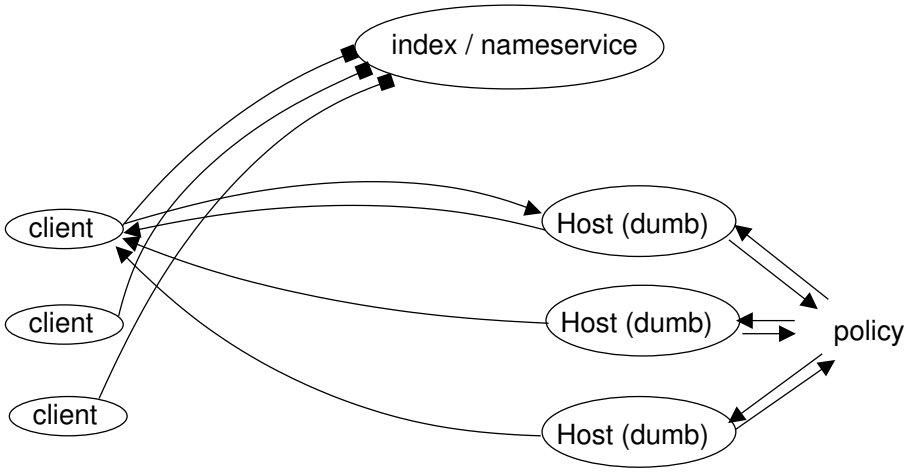

Figure 7.2: Firewall new.

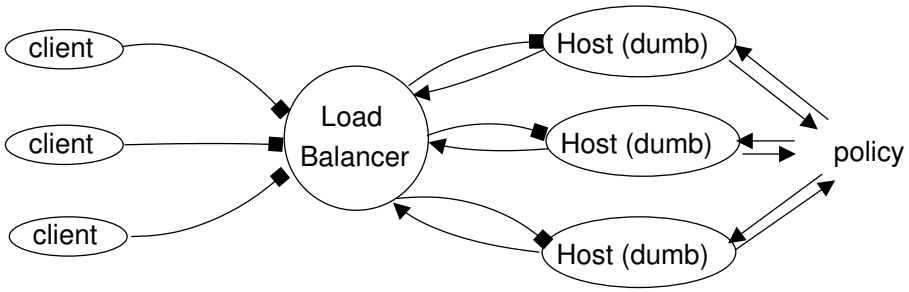

Figure 7.3: Load balancer old.

7.5 IPAM

7.6 Application centric infrastructure

7.7 Software defined datacentre and networking

7.8 WAN scaling methods and datacentre

BGP

7.8.1 Data consistency and transactional processing

7.8.2 QoS

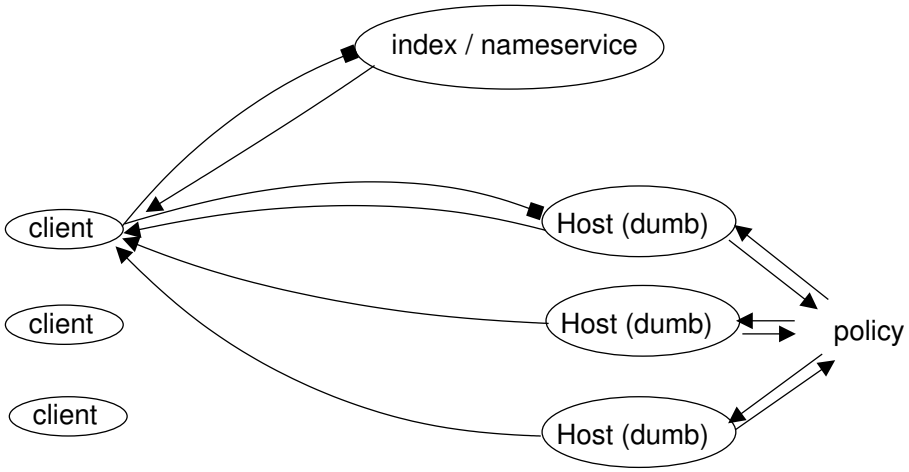

Figure 7.4: Load balancer new.

Bibliography

[Sny81] L. Snyder. Formal models of capability-based protection systems. *IEEE Transactions on Computers*, 30:172, 1981.