Язык SafeScript

Удобная настройка системы аварийного отключения оборудования

Почему не ZABBIX?

- Много кликов для запуска самой простой логики
- С другой стороны: не достаточно гибкая система задания логики при работе с большими кластерами (Ломоносов)
- Печальный опыт попытки выразить ТЗ на Ломоносов в терминах Заббикса

Альтернативный подход Clustrx.Safe

- Логика наблюдения за оборудованием может быть выражена в более гибких терминах специального языка
- Возможность выразить больше за меньшее время ценой ***небольшой*** инвестиции времени в изучение инструмента
- Язык позволяет начать думать про наблюдение за системой в более систематических терминах: событие, реакция, датчик и т д – т.е. то, что предусмотрено в языке. Это освобождает от сомнений при составлении последующих ТЗ

Язык SafeScript

- Основной инструмент задания логики – язык SafeScript
- Задает специальные понятия, характерные для предметной области: датчик, событие, реакция – ***ничего лишнего.***
- Если в будущем идти по пути упрощения – например, визуального программирования, язык все равно нужен, так как программа всегда сводится к набору инструкций, визуальные схемы переводятся в промежуточный язык.

Примеры

- Первое событие из ТЗ на Ломоносов
- Датчик опрашивается каждую секунду по SNMP

```
sensor(1).protocol(snmp).host(“10.10.1.223”).oid(OID).period(1),
```

- Событие проверяется каждые 5 секунд
- Источником события служит sensor(1)

```
event(1)
  .period(5)
  .source(sensor(1))
  .criteria(fun crit_event/1)
  .reaction(fun critical_reaction/1)
  .description("Срабатывание системы пожаротушения в зале М3"),

% показание датчика изменилось
crit_event_1(X) { C = change(X), (C <> 0) and (C <> undefined) }
```

Что можно?

- Считывать значения с датчиков по протоколам **IPMI**, **SNMP** (v2c), **MODBUS** + любые протоколы, которые можно считать внешним скриптом

- Выполнять реакцию по наступлению события. Реакция может быть произвольным скриптом.

- Отказоустойчивость: множество работников, но только один из них – главный, и только он выполняет реакцию.

Чего нельзя?

- Не хранит длинную историю по датчикам: только последние 100 значений. Остальное сохраняется средствами логирования.
- Нельзя заводить датчики динамически – это статическая информация, задаваемая с самого начала.
- Язык не является Turing-полным, т.е. сложные вычислительные алгоритмы не выражаются на этом языке. Для этой цели есть:
  - а) встроенные функции
  - б) внешние инструменты, к которым обращаемся через скрипт
- Вести наблюдение более чем за ~5000 датчиков одновременно с интервалом опроса датчиков 3 секунды. При бОльших потребностях можно установить два “ансамбля”, каждый из которых работает независимо от другого.

Пример посложнее

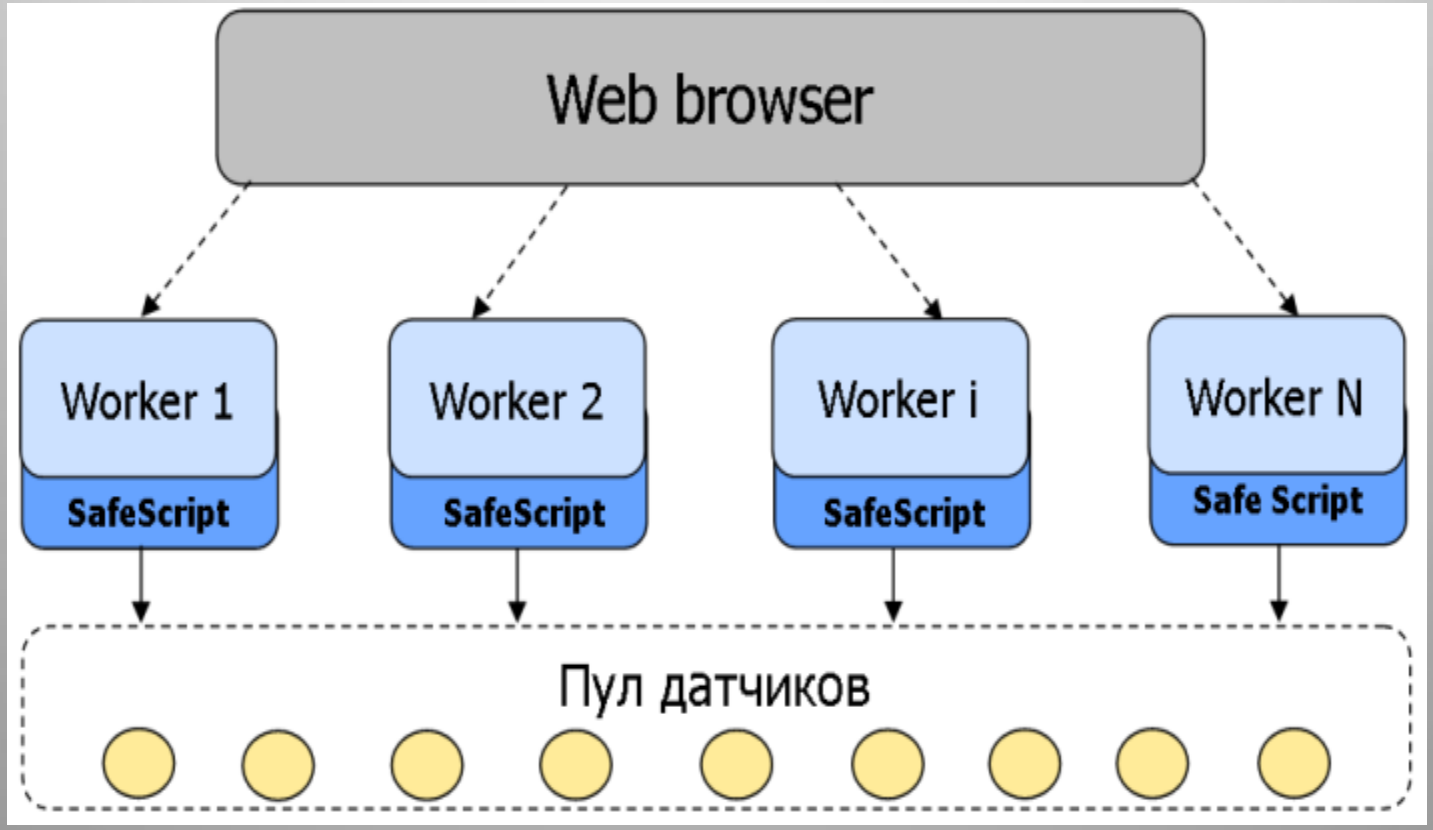
“Сложное” событие

```
sensor(2).protocol(snmp).host(IP1).oid(OID).period(1).emu_a1,
sensor(3).protocol(snmp).host(IP2).oid(OID).period(1).emu_a1,
…
sensor(N).protocol(snmp).host(IPn).oid(OID).period(1).emu_a1,

event(2).source(sensor.emu_a1).period(1).criteria(crit_2)
  .reaction(danger_reaction).description(“Пожар в коридоре 1”),

% если значение одного из датчиков отлично от нуля, то считаем событие
% произошедшим
crit_2(X) { sum(last(X)) <> 0 }

% реакция danger_reaction стандартная и поставляется вместе с системой (с версии 1.6)
danger_reaction(E) {
        Msg = “Произошло опасное событие “ ++ uid(E) ++ “ : “ ++ description(E),
        send_email(AdminEmail, Msg)
}
```

Язык SafeScript

- Центральное понятия языка – датчик, событие, тег и функция.
- Датчики определяют источники данных, события определяют действия, теги навешиваются на датчики для выбора из множества, функции используются для описания логики
- Функциональный язык с слабой динамической типизацией
  - 1 + “2.0” == 3.0 (True)
  - 1 + “A” –> exception
  - “2” + undefined == undefined
- Каждое заданное событие интерпретируется независимо от остальных в отдельном легковесном процессе (Erlang VM)
- События могут отслеживать состояния других событий через специальные функции (times_occured, last_at, active_period, is_active)

Как это работает?



Реализовано на Erlang

- Поддержка многопоточности на уровне языка
- Концепция supervisor: за каждым рабочим потоком стоит процесс, который перезапустит рабочий поток в случае его падения
- Использование функционального языка программирования способствует более простой отладке и пониманию приложения
- Не зависит от ОС: Erlang VM абстрагирует нас от нижележащей системы
- Сетевое взаимодействие поддерживается на уровне языка
- И при всём при этом – *быстрый (*скорость вычисления 10000 знаков числа PI в худшем случае уступает C++ не более, чем на порядок*)*
  - *http://shootout.alioth.debian.org/u32q/erlang.php*

Вопросы

Евгений Шишкин,
Компания Т-Платформы, отдел разработки ПО

Почта: evgeniy.shishkin@t-platforms.ru
Телефон: +7 (495) 959 5490 доб. 1124