# Distributed Systems — Power actions in a smart grid

Bart Wiegmans & Hessel van Apeldoorn

November 5, 2014

# Contents

# 1 Context/background

It is an unfortunate fact of electrical energy that it's production and consumption must be balanced at all times. Since the consumption of energy varies throughout the day as people and businesses employ machines and appliances network managers have always relied on various strategies to adapt production to the demand. The main strategy is to have a diverse array of power plants, some which can be started and stopped quickly (such as hydroelectrical and gas-fired power plants), and some which cannot adapt quickly but which are more economical in operation, such as coal-fired and nuclear power plants.

This strategy relies heavily on the use of fossil fuels, whose output is predictable. This poses new challenges in the coming century as fossil fuels become increasingly scarce and expensive. Moreover, the continued use of fossil fuels is likely to cause dangerous global climate change which, when it occurs, is certain to have large and disastrous impacts on natural ecosystems as well as human habitats and the economy. So to continue to use electrical energy - as any modern economy must - other, renewable resources need to be developed. Chief among these are wind energy and photovoltaic solar energy.

However, unlike fossil fuels the production of these energy from renewable sources cannot be adapted to meet demand. Energy is produced whenever the wind blows and the sun shines, even if we'd rather use it at any other time. At the same time energy production cannot be increased at times of great demand (such as during important sporting events). So the use of renewable energy brings about new challenges for the managers of the electricity grid. It is believed that this challenge can be met only by the implementation of so-called 'smart grid' systems.

The key idea of a smart grid is that the electricity consumers are made aware of and can react to (sudden) changes in the production capacity of the electricity grid. Just how such systems adapt to energy scarcity and abundance depends on the system. For example, a refrigerator which stays cool for a long time may switch off cooling at times of scarcity and compensate when electricity is abundant. For another example, modern computer systems contain sophisticated power management systems which allow them to vary energy use and performance nearly instantaneously. This adaptive capability may prove very valuable in combination with a smart grid system.

# 2 Problem statement

As the use of computing in modern societies has grown, so has the energy use devoted to computation. In 2010 this was measured to be 1of total human energy consumption [1]. More significantly, in the period between 2005 and 2010 the energy used by computing doubled, growing by 16,7% each year [1]. With the development of cloud computing, an increasing proportion of this energy is used in data centers. Given these trends, it becomes important to manage the energy consumption by data centers. Fortunately, because of the aforementioned sophisticated power management systems, this is practically doable.

We implemented a system designed to allow individual energy consumers (i.e. servers) in a data center to adapt to changes in energy availability. We assume that the relative abundance or scarcity of electrical energy is reflected in the minute-to-minute price of electricity expressed in eurocents per kWh, and that a single supplier sets this price. This implies some loss of generality since in the real world there may be multiple suppliers, each demanding a different price depending on the conditions of their respective producers. Moreover the actual cost of production depends on the amount produced - as more energy is required by the grid, more expensive power sources will be used to meet the demand. Such difficulties can be disregarded if we assume that energy producers - which set the price - assume the (financial) risk of having to tap into more expensive sources and that an energy broker process negotiates the best possible

deal between multiple energy producers, thereby providing the clients of this system with a single price. These negotiations between broker and producers are outside the scope of this project, though.

The clients of this system are the servers that can adapt their energy use individually. As servers in data centers are typically stacked in a rack, and racks need to be cooled as a unit, it might make more sense to adapt the energy use of a whole rack. In this case a client represents an entire rack of such blade servers. Clients are assumed to run jobs of varying economic value, and will change their energy consumption with the price depending on the value of their job and the job-specific relation between energy use and performance. For example, computationally intensive jobs will depend on CPU-performance (and energy use) more than a web crawling job will, since that process is likely to spend most time waiting for network requests. Thus each client will have it's own schedule for energy use and economic value and can adapt it's use accordingly. Note that with the development of cloud-hosted computing systems such as amazon web services, many 'jobs' in real-world data centers really do have an economic value assigned to them.

Although the nodes in this system sell and buy electrical energy in near-realtime, this system is assumed to act in an advisory, and not mandatory, fashion. That is to say, if a given node has agreed to buy 10kWh for a particular time period, and it really uses 11kWh, it is assumed that this discrepancy is penalized by paying a higher tariff. A client should not be shutdown for failing to negotiate power usage.

# 3 State of the Art

Energy consumption is an important issue for data centers. Data centers should always use energy as effective as possible. A metric to measure the effectiveness is the power usage effectiveness (PUE). This metric measures the amount of energy that was used as overhead compared to the energy used by servers. A smart energy grid should help to increase the PUE.

One of the companies that uses this metric for their data centers is Google[2]. Google states a few actions they take in order to use energy as efficiently as possible. These are: controlling air flow, temperature regulation and energy distribution optimization[3]. The last action is the one our project is focussing on. Energy is saved by optimizing AC/DC conversion for power to servers. Furthermore, redundant components are removed (e.g. slots for video cards).

A different approach is to use efficient Power Distribution Units (PDU)[4]. PDU's are placed just in front of a number of servers in a data center and efficiently distribute the energy across these servers. Rather than using a few large IT devices to distribute the power, thousands of PDU's are used to achieve a higher level of PUE. This approach lies closer to what we hope to improve with our project.

Besides losing as little energy as possible on transport to the server, it's also important to make sure that appropriate amounts of energy are sent to appropriate places. This can be achieved by using a smart energy grid. One company that has implemented this technology is HP[5]. The HP Data Center Smart Grid collects and communicates thousands of measurements across IT systems. This technique of HP is supposed to be used by IT managers. Based on the data collected, IT managers can manually optimize the power usage on a moment-to-moment basis.

Another interesting approach is taken by Power Assure[6]. Instead of focussing on optimizing energy distribution within a single data center, the application of Power Assure searches for the optimal energy distribution for a set of data centers. It makes use of the fact that power does not have the same price at a certain moment everywhere around the world. When a data center has to do a lot of computationally

expensive jobs and the power price is high at that moment, it may shift the jobs to another data center in the world where energy is cheaper at that particular moment.

# 4 Relation to Distributed Systems

Throughout our entire smart energy grid for data centers many techniques from the field of distributed systems are used. Even the data center itself can be considered as a distributed system, since it consists of several nodes all connected with each other through a network.

The most used technique is multicast, specifically ordered reliable multicast based on IP multicast. Nodes in a group have to reach concensus on the energy price together with the broker. In order to achieve this, the broker sends out a message using multicast to let hosts know the price of energy for a certain amount of energy for a certain amount of time. Nodes then try to come up with the amount of energy that they would like to buy. This is then multicast across all nodes for replication of the bought amounts.

Nodes are not always on. Whenever a node has nothing to do it may not be powered and connected to the network. We use dynamic host discovery to ensure that nodes can properly join the energy grid when they do need to do computations. A node knows which group it wants to join. It sends a message to the leader that it wants to join that particular group. The leader then acknowledges this join request and the entire group is notified of a new node entering the group.

Furthermore, it's also possible that nodes leave, including broker nodes that fulfill a leading position. In this case we need to assign a new broker. This is done by using the technique leader election. As soon as a node notices that the broker is down, a node will start an election given that the node itself can become the new broker. The election variant that we use in our energy grid is the Bully algorithm.

It may happen that nodes in the smart energy grid crash. This can occur due to hardware failure or human error, for example. There is thus the need to be able to solve crash faults as well as omission faults. In our project crash faults are solved by sending heartbeats. Which means that for every small period of time, a message is send to a node to check if it's still alive. Omission faults are present in sending messages to just one other node or multicasting it to all nodes within a group. In case of a failure of sending to one node, the node requests a resend of the message to the sender. In case of missing a multicast message, the entire group is asked to resend the message to the node that's missing a message.

# 5 Details of the technical implementation

Many aspects of our system are based on the raft consensus protocol [7]. However, there are several important differences. The main difference is that we use asynchronous IP multicast to deliver messages to the group of computers, whereas raft uses peer-to-peer remote procedure calls. This was done because raft is designed for a relatively small system (a clustor of 5), whereas our system is intended to serve a far greater number of clients. Most of the further differences from raft follow from this decision.

## 5.1 Reliable ordered messaging

Each process acquires a socket for receiving multicast messages, and a socket for receiving peer-to-peer messages. Both are unreliable, asynchronous UDP sockets. All messages are sent through the peer-to-peer socket, even those destined for the multicast group. This is possible due to the open nature of IP multicast, which allows any sender to send messages. As a result, messages send by any process to the multicast group are also delivered to that same process. This property simplifies the design because there

are fewer exceptions.

To ensure reliable delivery of peer-to-peer messages, all sent messages are kept in a buffer and need to be acknowledged by the receiver. Until they are acknowledged they will be sent periodically, stopping only when the failure of the receiver has been reported. Sequence numbers are used to distinguish different messages and to ensure ordering.

Multicast delivery on the other hand uses negative acknowledgements, or resend requests, to ensure eventual delivery to all hosts. Just as for peer-to-peer messages, a process issues sequence number on each multicast message it sends. On the receivers' end, this sequence number is used to ensure ordering using a holdback queue. Using only this sequence number, a receiving process guarantees delivery in sender order, but no more. When a process receives an out-of-order message, it is put into the holdback queue but not delivered. Periodically the process checks for gaps in the multicast holdback queue and sends a resend request. If the original sender is still alive, the request is sent directly to this host, which will retransmit the message. If the host has failed, the request is sent to the group. Whenever a multicast message has been delivered, a process retains it. So if any host has delivered a given multicast message, all other hosts can still request retransmission if the original sender has died.

## 5.2 Heartbeat Messages

Resend requests only guarantee delivery as long as all hosts eventually learn of messages that they have missed. For this and other purposes the system leader regularly sends 'heartbeat' messages informing other hosts about the state of system. Each heartbeat message contains the sequence number of the last delivered multicast messsage known to the leader. When a host receives such a message, it can check if it has sent a more recent message itself, or if it has delivered a more recent message from another host that is now dead. Represented in pseudocode this algorithm is as follows:

```
for senderPid, sequenceNr in heartbeatState:
  if senderPid == myPid and sequenceNr > lastSendMessage.sequenceNr:
      resendMessagesSince(sequenceNr)
  else if isDead(senderPid) and sequenceNr < lastDeliveredMessage(senderPid).sequenceNr:
      resendMessagesOf(senderPid, sequenceNr)
```

Because a leader is always eventually elected and all crash failures are eventually noticed and relayed, this ensures the eventual delivery of all messages, as long as at least one of the hosts that delivered the message is alive and reachable.

Ordered messages that are sent by an unknwon host are dropped, because otherwise they could be delivered multiple times. Thus, a host must be known before its (reliable) messages can be delivered. Because of this, some messages are not sent reliably, such as heartbeat messages and lifesign responses. The application can function quite well if some of these messages are lost. Moreover, the utility of these is liimited by their timely delivery - an eventually-delivered heartbeat message is no good if more recent heartbeats have been delivered since.

## 5.3 Host discovery and failure detection

The open group property of IP multicast is used to facilitate host discovery. We use an IP number in the range allocated for local use, namely 224.0.0.224. When a host connects to the network, it connects to the IP multicast group and waits for several heartbeat periods to listen for heartbeat messages. When it receives a heartbeat it assumes the sender to be its leader and replies with a lifesign message.

The group leader in turn is responsible for managing the hosts in the shared group. When it receives a lifesign message from a unknown host, it assigns a process id (PID) to the host, and multicasts a 'join' message to the group. This join message contains the newly assigned PID, as well as the address of the host, and is sent using reliable multicasting. It also sends a 'welcome' message to the new host, informing it of its new PID. This is necessary because hosts do not, in principle, know their own network address. A host may be behind one or more NAT systems, and the address that is known to other hosts may be quite different from the address by which it knows itself. Note that the externally visible address of any host must be unique within the network, otherwise the host cannot be uniquely addressed. So each host is eventually assigned a unique PID, and each host recognises another by mapping their externally visible network address to the leader-assigned PID. This PID is also used to address a message to any other node.

When a known host fails to respond to heartbeat messages for a given period of time - a multiple of the heartbeat period of the system - the leader notices, and sends a 'leave' message to the group telling each host to remove the host from the group. A PID is assigned only once, so if a new process is connected with the same network address as an earlier process it will be assigned a new, unique PID.

Consensus over the hosts in this group is reached, in principle, by the reliable, ordered delivery of all join and leave messages sent by all leaders.

## 5.4 Bootstrapping and leader election

As mentioned above, the system can only deliver reliable messages to a host it already knows, from a host that receiver also knows. This poses a special challenge for leader election, since without a leader no attempts are made to determine who belongs to the group and who does not. Our solution is to use unreliable messaging for leader election (and hope for the best).

Inspired by the raft consensus algorithm, we use randomized election timeouts to decrease the chances that two nodes try to elect themselves at the same time. An election is started when a host detects that no heartbeat messages have been received for a certain amount of time (a multiple of the heartbeat period, similar to the timeout used to detect host failures), The host then mulitcasts a 'vote request' message requesting other hosts to vote for itself. If after another timeout it has received a majority of votes, it declares itself leader and starts sending out heartbeat messages. The sending of heartbeat messages signals to other nodes that is hasb ecome a leader. On the other hand, if it receives a new heartbeat message before that time the election is canceled.

Election (and leadership) rounds are divided into terms, which are monotonically incrementing numbers. Before a host sends out vote request messages it increases its current election term. A host only replies positively to a vote request if the election term of the requester is equal or greater than its own election term and if it hasn't voted for any other node during the same term.

Because the vote request and vote reply messages are sent unreliably, omission failures may cause only a subset of hosts to receive the initial request, and only a subset of the replies may be seen by the requester. Alarming as this may seem, this is not really a problem. Before the election starts, an election candidate takes the count of the group size - as far as it knows it at that time - and uses that as an estimate for the real group size. It will not declare itself leader unless it has received a majority vote compared to the maximum of this group size estimate and the total votes it actually receives. Thus, even though messages may be lost, a host cannot declare itself leader unless it actually received a majority vote from the group. The exception is the initial case of network setup, in which case the size of the network is not known beforehand. Using the size of the estimated group also means that if a network split occurs and no host

can reach a majority of the former group, no host can elect itself. This is intentional, as no consensus can be expected to be reached in that case.

### 5.5 Power auctions

The group leader automatically assumes the role of power broker. The price is set using a random walk algorithm that periodically computes a new energy price, which is then multicast to the group. Clients then compute the amount of energy they wish to buy, based upon the price of energy, the economic value of their job, and the relative value of more energy (and more computational power) to the completion of their job. This amount is then multicast back to the group, which replicates the decisions of all hosts. It is assumed that the broker will use this amount for continued negotiations.

Because the auctions are relevant for a particular time period they are marked with a specific time and duration. To compensate for clock skew each heartbeat message carries with it the time it was sent according to the leader. With this information a client can compute the average offset between its clock and that of the leader and accurately estimate when this time period actually occurs.

### 5.6 Failure modes

Using heartbeat and lifesign messages, the system can identify and tolerate a crash failure, incuding failure of the leader node. Channel ommissions are dealt with using message resending and acknowledgements. Message reordering and duplicate deliveries also dealt with correctly using a message sequencer and a holdback queue.

What happens in case of a catastrophic network split depends on which side of the split the majority resides. In case the old leader is split from the majority a second leader will be elected, which will then proceed in dropping the hosts on the other side of the split. The old leader will not step down, however when the network is restored the second leader will take over since it must have been elected in a later term. The second leader will then proceed in giving hosts in the newly reconnected group new PIDs, since it will have assumed these nodes dead. Note that network splits are common events in data centers[8].

Because the system relies on resending of third-party messages for guaranteeing delivery, it is very sensitive to byzantine failures or attacks. A possible solution to this - indeed a sensible precaution, given that the messages transmitted carry a financial obligation with them - is to sign the cryptographically messages using a public/private key pair. This would prohibit message forgery.

Using negative acknowledgments / resend requests, multicast delivery is not entirely guaranteed, as is demonstrated by the following chain of events:

- Node A multicasts message A:1
- Only Node B receives message A:1.
- Node A crashes.
- Node B multicasts message B:1, which is eventually delivered to all nodes via resending
- Node B then crashes

If node B had crashed immediately after receiving A:1 (before sending B:1), we could have claimed some form of consistency, since no *living* nodes ever delivered A:1. However because node B was able to send B:1, we essentially have a delivered message B:1 without it's causal predecessor. The system doesn't

promise causually ordered delivery, however in some cases it would have been useful.

A clear example where causual ordering would have been useful is group management in conditions of leader change. Suppose the following sequence of events would occur:

- Node X and Y both join the network at the same time.
- Leader A adds node X under PID 17 (message A:100). Node B does not initially deliver this message, but node C does.
- Leader A crashes.
- Node B is elected leader.
- Node B adds node Y under PID 17 (B:88), and node X under PID 18 (B:89).
- Node B sends out heartbeat messages, causing node C to resend Leader A's message (A:100) that added node X under PID 17.
- To node B, Node X now occupies both PID 17 and PID 18, and node Y does not have a PID.
- This causes node B to assign Node Y PID 19.
- Node D, which already delivered A:100, is not effected by the retransmission of A:100, which causes it to think node X has PID 18, and now node Y has PID 19 as well as 17.
- Since Node B thinks that the lifesign messages from node X belong to PID 17 (which was last added in node B's address-to-pid map), so it will eventually drop PID 18 (B:90).
- This will make node D think that node X is dead, which it is not.

Causally ordered messages would not prohibit this sequence of events to occur, since the sending of A:100 did not happen-before the sending of either B:88 or B:89. However, having the vector clocks of these messages would allow the group management routines to determine, in case of a leader change, whether the join and leave messages causually follow each other. Combined with a message log this would allow the group to roll back to a state consistent with that of the new leader.

A radically different solution, rather than having a single leader assign PIDs to new hosts, would be to have nodes assign a (randomly generated) PID to themselves. New hosts would then repeatedly announce a their pressence with a new PID until the group leader verifies that it is indeed unique, at which point the new host is considered to be added to the group. Alternatively, the public key mentioned above, necessary to verify signed messages, could be used as a unique process identifier.

Node identities are not persisted across crashes, which is unrealistic for a system which intends to deal in financial matters. Again, a persistent public key is a far more realistic and practical means of identification for individual nodes, given that it also protects against message forgery.

## References


[1] Jonathan G. Koomey. URL http://www.koomey.com/post/8323374335. Last access on Nov. 5th, 2014.

[2] Google, . URL http://www.google.com/about/datacenters/efficiency/internal/. Last access on Nov. 5th, 2014.

[3] Google, . URL http://www.google.com/about/datacenters/efficiency/external/index.html#best-practices. Last access on Nov. 5th, 2014.

[4] Neil Rasmussen and Wendy Torell. URL http://www.apcmedia.com/salestools/WTOL-7ANTKY/WTOL-7ANTKY_R3_EN.pdf. Last access on Nov. 5th, 2014.

[5] HP. URL http://www8.hp.com/us/en/hp-information/environment/hp-data-center-smart-grid.html#.VFqS3XVdWkA. Last access on Nov. 5th, 2014.

[6] Jeff St. John. URL http://www.greentechmedia.com/articles/read/putting-the-smart-grid-enabled-data-center-to-the-test. Last access on Nov. 5th, 2014.

[7] Raft. URL http://raftconsensus.github.io/. Last access on Nov. 5th, 2014.

[8] Jepsen. URL http://aphyr.com/posts/288-the-network-is-reliable. Last access on Nov. 5th, 2014.