APPENDIX

Cloud projects

A

Several projects for students enrolled in a cloud computing class and a research project involving an extensive simulation are discussed in this section. These projects reflect the great appeal of cloud computing for various types of applications. Large-scale simulation of complex systems, such as the one discussed in Sections A.1 and A.3, can only be done using the large pool of resources provided by clouds. Cloud services, such as the one discussed in Sections A.2 and A.4, are another important class of applications.

Design projects in which multiple alternatives must be evaluated and compared, such as the one discussed in Section A.5, benefit from a cloud environment. Several, possibly many, design alternatives can be simulated concurrently; then, the selection of the best alternatives based on different performance objectives can be done by multiple instances running concurrently. High-performance computing applications such as the one discussed in Section A.6 require resources that can only be provided by supercomputers or computer clouds. Clouds are better alternatives than supercomputers due to easier access and lower cost. Applications of machine learning and healthcare applications take advantage of GPUs and TPUs available in some AWS instances. A simulation study of ML algorithm scalability, a cloud-based task management system, and a cloud-based health-monitoring application are discussed in Sections A.7, A.8, and A.9, respectively.

A.1 Cloud simulation of a distributed trust algorithm

Cloud-based simulation for trust evaluation in a Cognitive Radio Networks (CRN) [63] is discussed first. The available communication spectrum is a precious commodity, and the objective of a CRN is to use the communication bandwidth effectively, while attempting to avoid interference with licensed users. Two main functions necessary for the operation of a CRN are spectrum sensing and spectrum management; the former detects unused spectrum, and the latter determines the optimal use of the available spectrum. Spectrum sensing in CRNs is based on information provided by the nodes of the network. The nodes compete for the free channels, and some may supply deliberately distorted information to gain advantage over the other nodes; thus, trust determination is critical for the management of CRNs.

Cognitive radio networks. Research in the last decade reveals a significant temporal and spatial underutilization of the allocated spectrum, thus the motivation to opportunistically harness the vacancies of spectrum at a given time and place.

The original goal of cognitive radio, first proposed at Bell Labs [351,352], was to develop a software-based radio platform that allows a reconfigurable wireless transceiver to automatically adapt

its communication parameters to network availability and to user demands. Today, the focus of cognitive radio is on spectrum sensing [76].

We recognize two types of devices connected to a CRN, primary and secondary; *primary* devices have exclusive rights to specific regions of the spectrum, while *secondary* devices enjoy dynamic spectrum access and are able to use a channel, provided that the primary, licensed to use that channel, is not communicating. Once a primary starts its transmission, the secondary using the channel is required to relinquish it and identify another free channel to continue its operation; this mode of operation is called an *overlay mode.*

Cognitive radio networks are often based on *cooperative spectrum-sensing* strategy. In this mode of operation, each device determines the occupancy of the spectrum based on its own measurements combined with information from its neighbors and then shares its own spectrum occupancy assessment with its neighbors [186,463,464].

Information sharing is necessary because a device alone cannot determine true spectrum occupancy. A secondary device has a limited transmission and reception range; device mobility combined with wireless channel impairments, such as multipath fading, shadowing, and noise, add to the difficulties of gathering accurate information by a single device.

Individual devices of a centralized, or infrastructure-based, CRN send the results of their measurements regarding spectrum occupancy to a central entity, whether a base station, an access point, or a cluster head. This entity uses a set of *fusion rules* to generate the spectrum occupancy report and then distributes it to the devices in its jurisdiction. The area covered by such networks is usually small because global spectrum decisions are affected by the local geography. There is another mode of operation based on the idea that a secondary device operates at a much lower power level than a primary one. In this case, the secondary can share the channel with the primary as long as its transmission power is below a threshold, μ, that has to be determined periodically. In this scenario, the receivers wishing to listen to the primary are able to filter out the "noise" caused by the transmission initiated by secondaries if the signal-to-noise ratio (S/N) is large enough.

We are only concerned with the overlay mode whereby a secondary device maintains an *occupancy report* that gives a snapshot of the current status of the channels in the region of the spectrum it is able to access. The occupancy report is a list of all the channels and their states, e.g., 0 if the channel is free for use and 1 if the primary is active. Secondary devices continually sense the channels they can access to gather accurate information about available channels.

The secondary devices of an ad hoc CRN compete for free channels, and the information one device may provide to its neighbors could be deliberately distorted; malicious devices will send false information to the fusion center in a centralized CRN. Malicious devices could attempt to deny the service or to cause other secondary devices to violate spectrum allocation rules. To *deny the service,* a device will report that free channels are used by the primary. To entice the neighbors to commit FCC violations, the occupancy report will show that channels used by the primary are free. This attack strategy is called *secondary spectrum-data falsification (SSDF)* or a Byzantine attack.[1] Thus, trust determination is a critical issue for CR networks.

Trust. The actual meaning of *trust* is domain and context specific. Consider for example networking; at the MAC-layer (Medium Access Control) multiple-access protocols assume that all senders follow

[1] See Section 10.13 for a brief discussion of Byzantine attacks.

the channel access policy, e.g., in CSMA-CD, a sender senses the channel and then attempts to transmit if no one else does. In a store-and-forward network, trust assumes that all routers follow a best-effort policy to forward packets towards their destination. We shall use node instead of device throughout the remaining of this section.

In the context of cognitive radio, trust is based on the quality of information regarding channel activity provided by a node. The status of individual channels can be assessed by each node, based on the results of its own measurements combined with information provided by its neighbors, as is the case of several algorithms discussed in the literature [103,463].

The alternative discussed in Section A.2 is to have a cloud-based service that collects information from individual nodes, evaluates the state of each channel based on the information received, and supplies this information on demand. Evaluation of the trust and identification of untrustworthy nodes are critical for both strategies [390].

A distributed algorithm for trust management in cognitive radio. The algorithm computes the trust of node $1 \leq i \leq n$ in each node in its vicinity, $j \in V_i$, and requires several preliminary steps. The basic steps executed by a node i at time t are:

1. Determine node i's version of the occupancy report for each one of the K channels:

$$S_i(t) = \{s_{i,1}(t), s_{i,2}(t), \ldots, s_{i,K}(t)\}. \tag{A.1}$$

   In this step node i measures the power received on each of the K channels.
2. Determine the set $V_i(t)$ of the nodes in the vicinity of node i. Node i broadcasts a message, and individual nodes in its vicinity respond with their *NodeId*.
3. Determine the distance to each device $j \in V_i(t)$ using the algorithm described in this section.
4. Infer the power as measured by each device $j \in V_i(t)$ on each channel $k \in K$.
5. Use the location and power information determined in the previous two steps to infer the status of each channel:

$$s^{infer}_{i,k,j}(t) \quad \text{with} \quad 1 \leq k \leq K, \;\; j \in V_i(t). \tag{A.2}$$

   A secondary node j should have determined: 0 if the channel is free for use, 1 if the primary device is active, and X if it cannot be determined:

$$s^{infer}_{i,k,j}(t) = \begin{cases} 0 & \text{if secondary node j decides that channel k is free} \\ 1 & \text{if secondary node j decides that channel k is used by the primary} \\ X & \text{if no inference can be made.} \end{cases} \tag{A.3}$$

6. Receive the information provided by neighbor $j \in V_i(t)$, $S^{recv}_{i,k,j}(t)$.
7. Compare the information provided by neighbor $j \in V_i(t)$

$$S^{recv}_{i,k,j}(t) = \{s^{recv}_{i,1,j}(t), s^{recv}_{i,2,j}(t), \ldots, s^{recv}_{i,K,j}(t)\} \tag{A.4}$$

   with the information inferred by node i about node j

$$S^{infer}_{i,k,j}(t) = \{s^{infer}_{i,1,j}(t), s^{infer}_{i,2,j}(t), \ldots, s^{infer}_{i,K,j}(t)\}. \tag{A.5}$$

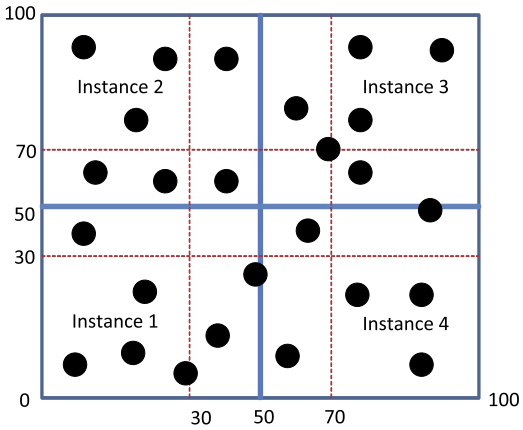


FIGURE A.1

Data partitioning for the simulation of a trust algorithm; the area covered is of size 100×100 units. The nodes in the four subareas of size 50×50 units are processed by an instance of the cloud application. The subareas allocated to an instance overlap to enable an instance to have all the information about a node in its coverage area.

8. Compute the number of matches, mismatches, and cases when no inference is possible, respectively:

$$\alpha_{i,j}(t) = \mathcal{M}\left[S^{infer}_{i,k,j}(t), S^{recv}_{i,k,j}(t)\right], \tag{A.6}$$

with $\mathcal{M}$ being the number of matches between the two vectors,

$$\beta_{i,j}(t) = \mathcal{N}\left[S^{infer}_{i,k,j}(t), S^{recv}_{i,k,j}(t)\right], \tag{A.7}$$

with $\mathcal{N}$ being the number of mismatches between the two vectors, and $X_{i,j}(t)$ being the number of cases where no inference could be made.

9. Use the quantities $\alpha_{i,j}(t)$, $\beta_{i,j}(t)$, and $X_{i,j}(t)$ to assess the trust in node j. For example, compute the trust of node i in node j at time t as:

$$\zeta_{i,j}(t) = \left[1 + X_{i,j}(t)\right] \frac{\alpha_{i,j}(t)}{\alpha_{i,j}(t) + \beta_{i,j}(t)}. \tag{A.8}$$

Simulation of the distributed-trust algorithm. The cloud application is a simulation of a CRN to assess the effectiveness of a particular trust-assessment algorithm. Multiple instances of the algorithm run concurrently on an AWS cloud. The area where the secondary nodes are located is partitioned in several overlapping subareas as in Fig. A.1. The secondary nodes are identified by an instance Id, *iId*, as well as a global Id, *gId*. The simulation assumes that the primary nodes cover the entire area; thus their position is immaterial.

The simulation involves a controller and several cloud instances; in its initial implementation, the controller runs on a local system under Linux Ubuntu 10.04 LTS. The controller supplies the data, the

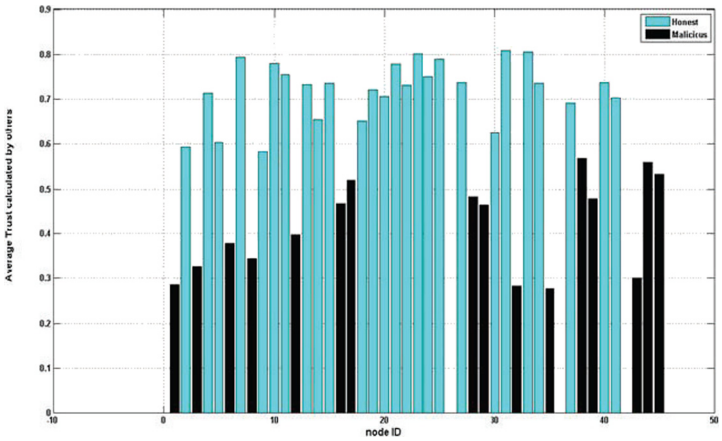


FIGURE A.2

The trust values computed using the distributed-trust algorithm. The secondary nodes programmed to act maliciously have a trust value less than 0.6 and many less than 0.5, lower than that of the honest nodes.

trust program, and the scripts to the cloud instances; the cloud instances run under the Basic 32-bit Linux image on AWS, the so-called *t1.micro*. The instances run the actual trust program and compute the instantaneous trust inferred by a neighbor; the results are then processed by an *awk*[2] script to compute the average trust associated with a node as seen by all its neighbors. On the next version of the application, the data is stored on the cloud using the *S3* service, and the controller also runs on the cloud.

In the simulation discussed here, the nodes with

$$gId = \{1, 3, 6, 8, 12, 16, 17, 28, 29, 32, 35, 38, 39, 43, 44, 45\} \tag{A.9}$$

were programmed to be dishonest. The results show that the nodes programmed to act maliciously have a trust value lower than that of the honest nodes; their trust value is always lower than 0.6 and, in many instances, lower than 0.5; see Fig. A.2. We also observe that the node density affects the accuracy of the algorithm; the algorithm predicts more accurately the trust in densely populated areas. As expected, nodes with no neighbors are unable to compute the trust. In practice, node density is likely to be nonuniform, high density in a crowded area such as a shopping mall and considerably lower in surrounding areas. This indicates that, when trust is computed using information provided by all secondary nodes, we can expect a higher accuracy of trust determination.

[2] The AWK utility is based on a scripting language and used for text processing; in this application, it is used to produce formatted reports.

A.2 A trust management service

The cloud service discussed in this section [63] is an alternative to the distributed-trust management scheme analyzed in Section A.1. Mobile devices are ubiquitous nowadays and their use will continue to increase. Clouds are emerging as the computing and the storage engines of the future for a wide range of applications. There is a symbiotic relationship between the two: Mobile devices can consume, as well as produce, very large amounts of data, while computer clouds have the capacity to store and deliver such data to the user of a mobile device. To exploit the potential of this symbiotic relationship, we propose a new cloud service for the management of wireless networks.

Mobile devices have limited resources. While new generations of smart phones and tablet computers are likely to use multicore processors and have a fair amount of memory, power consumption is still, and will continue to be in the near future, a major concern. Thus, it seems reasonable to delegate compute- and data-intensive tasks to the cloud.

Transferring computations related to CRN management to a cloud supports the development of new, possibly more accurate, resource management algorithms. For example, algorithms to discover communication channels currently in use by a primary transmitter could be based on past history but are not feasible when the trust is computed by the mobile device. Such algorithms require massive amounts of data and can also identify malicious nodes with high probability.

Mobile devices such as smartphones and tablets are able to communicate using two networks: (i) a cellular wireless network; and (ii) a WiFi network. The service we propose assumes that a mobile device uses the cellular wireless network to access the cloud, while the communication over the WiFi channel is based on cognitive radio (CR). The amount of data transferred using the cellular network is limited by the subscriber's data plan, but no such limitation exists for the WiFi network. The cloud service discussed next will allow mobile devices to use the WiFi communication channels in a cognitive radio network environment and will reduce the operating costs for the end-users.

While the focus of our discussion is on trust management for CRN networks, the cloud service we propose can be used for tasks other than the bandwidth management. For example, routing in a mobile ad hoc network, detection, and isolation of noncooperative nodes, and other network management and monitoring functions could benefit from the identification of malicious nodes.

Model assumptions. The cognitive radio literature typically analyzes networks with a relatively small number of nodes active in a limited geographic area; thus, all nodes in the network sense the same information on channel occupancy. Channel impairments such as signal fading, noise, and so on, cause errors and lead trustworthy nodes to report false information. We consider networks with a much larger number of nodes distributed over a large geographic area; because the signal strengths decays with the distance, we consider several rings around a primary tower. We assume a generic fading model given by the following expression:

$$\gamma_k^i = T_k \times \frac{A^2}{s_{ik}^{\alpha}}, \tag{A.10}$$

where γ_k^i is the received signal strength on channel k at location of node i, A is the frequency constant, $2 \leq \alpha \leq 6$ is path loss factor, s_{ik}^{α} is the distance between primary tower P_k and node i, and T_k is the transition power of primary tower P_k transmitting on channel k.

In our discussion, we assume that there are K channels labeled $1, 2, \ldots, K$ and that the primary transmitter P^k transmits on channel k. The algorithm is based on several assumptions regarding the

secondary nodes, the behavior of malicious nodes, and the geometry of the system. First, we assume the following:

- The secondary modes are mobile devices; some are slow-moving, while others are fast-moving.
- They cannot report their position because they are not equipped with a GPS system.
- The clocks of the mobile devices are not synchronized.
- The transmission and reception range of a mobile device can be different.
- The transmission range depends on the residual power of each mobile device.

We assume that the malicious nodes in the network are a minority and their behavior is captured by the following assumptions:

- The misbehaving nodes are malicious, rather than selfish; their only objective is to hinder the activity of other nodes whenever possible, a behavior distinct from the one of selfish nodes motivated to gain some advantage.
- The malicious nodes are uniformly distributed in the area we investigate.
- The malicious nodes do not collaborate in their attack strategies.
- The malicious nodes change the intensity of their Byzantine attack in successive time slots; similar patterns of malicious behavior are easy to detect, and an intelligent attacker is motivated to avoid detection.

The geometry of the system is captured by Fig. A.3. We distinguish primary and secondary nodes and the cell towers used by the secondary nodes to communicate with the service running on the cloud.

We use a majority voting rule for a particular ring around a primary transmitter; the global decision regarding the occupancy of a channel requires a majority of the votes. Since the malicious nodes are a minority and they are uniformly distributed, the malicious nodes in any ring are also a minority; thus, a ring-based majority fusion is representative of accurate occupancy for the channel associated with the ring.

All secondary nodes are required to register first and then to transmit periodically their current power levels, as well as their occupancy report for each one of the K channels. As mentioned in the introductory discussion, the secondary nodes connect to the cloud using the cellular network. After a mobile device is registered, the cloud application requests the cellular network to detect its location; the towers of the cellular network detect the location of a mobile device by triangulation with an accuracy that is a function of the environment and is of the order of 10 m. The location of the mobile device is reported to the cloud application every time they provide an occupancy report.

The nodes that do not participate in the trust computation will not register in this cloud-based version of the resource management algorithm, thus they do not receive the occupancy report and cannot use it to identify free channels. Obviously, if a secondary node does not register, it cannot influence other nodes and prevent them from using free channels, or tempt them to use busy channels.

In the registration phase, a secondary node transmits its MAC address, and the cloud responds with the tuple (Δ, δ_s). Here, Δ is the time interval between two consecutive reports, chosen to minimize the communication and the overhead for sensing the status of each channel. To reduce the communication overhead, secondary nodes should transmit only the changes from the previous status report. $\delta_s < \Delta$ is the time interval to the first report expected from the secondary node. This scheme provides a pseudo-synchronization, so that the data collected by the cloud and used to determine the trust is based on observations made by the secondary nodes at about the same time.

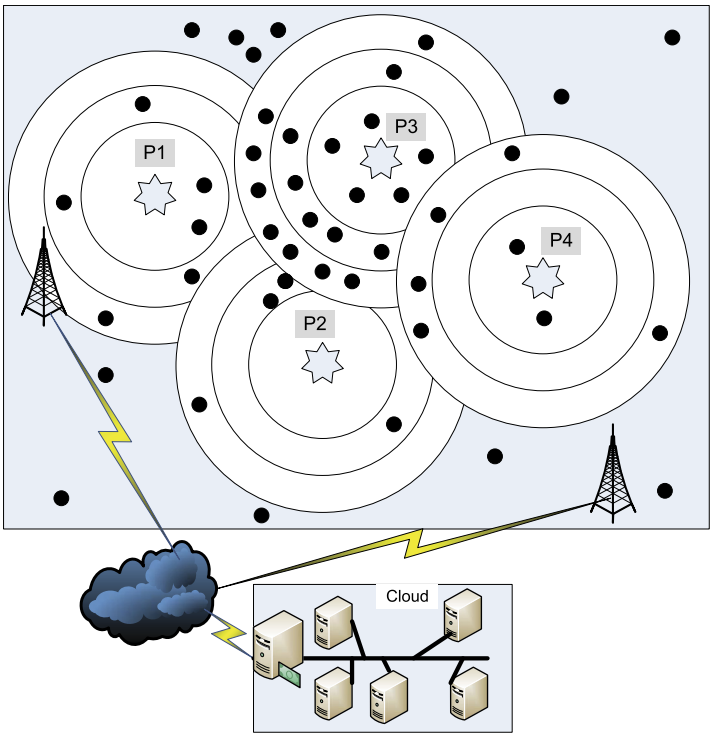


FIGURE A.3

Schematic representation of a CR layout; four primary nodes, P_1–P_4, a number of mobile devices, two towers for a cellular network, and a cloud are shown. Not shown are the hotspots for the WiFi network.

An algorithm for trust evaluation based on historical information. The cloud computes the probable distance d_i^k of each secondary node i from the known location of a primary transmitter, P^k. Based on signal attenuation properties, we conceptualize N circular rings centered at the primary where each ring is denoted by $\mathcal{R}_r^k$, with $1 \leq r \leq N$ the ring number.

The radius of a ring is based on the distance d_r^k to the primary transmitter P^k. A node at a distance $d_i^k \leq d_1^k$ is included in the ring $\mathcal{R}_1^k$, nodes at distance $d_1^k < d_i^k \leq d_2^k$ are included in the ring $\mathcal{R}_2^k$, and so on. The closer to the primary, the more accurate the channel occupancy report of the nodes in the ring should be. Call n_r^k the number of nodes in ring $\mathcal{R}_r^k$.

At each report cycle at time t_q, the cloud computes the occupancy report for channel $1 \leq k \leq K$ used by primary transmitter P^k. The status of channel k reported by node $i \in \mathcal{R}_r^k$ is denoted as $s_i^k(t_q)$. Call $\sigma_{one}^k(t_q)$ the count of the nodes in the ring $\mathcal{R}_r^k$ reporting that the channel k is not free (reporting $s_i^k(t_q) = 1$) and $\sigma_{zero}^k(t_q)$ the count of those reporting that the channel is free (reporting $s_i^k(t_q) = 0$):

$$\sum_{one}^{k}(t_q) = \sum_{i=1}^{n_r^k} s_i^k(t_q) \quad \text{and} \quad \sigma_{zero}^k(t_q) = n_r^k - \sigma_{one}^k(t_q). \tag{A.11}$$

Then, the status of channel k reported by the nodes in the ring R_r^k is determined by majority voting as:

$$\sum_{R_r}^{k}(t_q)\begin{cases}=1 & \text{when } \sum_{one}^{k}(t_q) \geq \sum_{zero}^{k}(t_q)\\ =0 & \text{otherwise}\end{cases}. \tag{A.12}$$

To determine the trust in node i, we compare $s_i^k(t_q)$ with $\sigma_{R_r}^k(t_q)$; call $\alpha_{i,r}^k(t_q)$ and $\beta_{i,r}^k(t_q)$ the number of matches and, respectively, mismatches in this comparison for each node in the ring R_r^k. We repeat this procedure for all rings around P^k and construct

$$\alpha_i^k(t_q) = \sum_{r=1}^{n_r^k} \alpha_{i,r}^k(t_q) \quad \text{and} \quad \beta_i^k(t_q) = \sum_{r=1}^{n_r^k} \beta_{i,r}^k(t_q). \tag{A.13}$$

Node i will report the status of the channels in the set $C_i(t_q)$, the channels with index $k \in C_i(t_q)$; then, quantities $\alpha_i(t_q)$ and $\beta_i(t_q)$ with $\alpha_i(t_q) + \beta_i(t_q) =\mid C_i(t_q) \mid$ are

$$\alpha_i(t_q) = \sum_{k \in C_i} \alpha_i^k(t_q) \quad \text{and} \quad \beta_i(t_q) = \sum_{k \in C_i} \beta_i^k(t_q). \tag{A.14}$$

Finally, the global trust in node i is a random variable

$$\zeta_i(t_q) = \frac{\alpha_i(t_q)}{\alpha_i(t_q) + \beta_i(t_q)}. \tag{A.15}$$

The trust in each node at each iteration is determined using a strategy similar to the one discussed earlier; its status report, $S_j(t)$, contains only information about the channels it can report on and only if the information has changed from the previous reporting cycle.

Then, a statistical analysis of the random variables for a window of time W, $\zeta_j(t_q), t_q \in W$ allows us to compute the moments and a 95% confidence interval. Based on these results, we assess if node j is trustworthy and eliminate the untrustworthy ones when we evaluate the occupancy map at the next cycle. We continue to assess the trustworthiness of all nodes and may accept the information from node j when its behavior changes.

Let us now discuss the use of historical information to evaluate trust. We assume a sliding window $W(t_q)$ consists of n_w time slots of duration τ. Given two decay constants k_1 and k_2, with $k_1 + k_2 = 1$, we use an exponential averaging giving decreasing weight to old observations. We choose $k_1 \ll k_2$ to give more weight to the past actions of a malicious node. Such nodes attack only intermittently and try to disguise their presence with occasional good reports; the misbehavior should affect the trust more than the good actions. The history-based trust requires the determination of the two quantities

$$\alpha_i^H(t_q) = \sum_{i=0}^{n_{w-1}} \alpha_i(t_q - i\tau)k_1^i \quad \text{and} \quad \beta_i^H(t_q) = \sum_{i=0}^{n_{w-1}} \beta_i(t_q - i\tau)k_2^i. \tag{A.16}$$

Then, the history-based trust for node i valid only at times $t_q \geq n_w\tau$ is

$$\zeta_i^H(t_q) = \frac{\alpha_i^H(t_q)}{\alpha_i^H(t_q) + \beta_i^H(t_q)}. \tag{A.17}$$

For times $t_q < n_w \tau$, the trust will be based only on a subset of observations rather than a full window based on n_w observations.

This algorithm can also be used in regions where the cellular infrastructure is missing. An ad hoc network could allow the nodes that cannot connect directly to the cellular network to forward their information to nodes closer to the towers and then to the cloud-based service.

Simulation of the history-based algorithm for trust management. The aim of the history-based trust evaluation is to distinguish between trustworthy and malicious nodes. We expect the ratio of malicious to trustworthy nodes and the node density to play an important role in this decision. The node density ρ is the number of nodes per unit of area. In our simulation experiments, the size of the area is constant, but the number of nodes increases from 500 to 2 000; thus the node density increases by a factor of four. The ratio of the number of malicious to the total number of nodes varies between $\alpha = 0.2$ to a worst case of $\alpha = 0.6$.

The performance metrics we consider are: the average trust for all nodes, the average trust of individual nodes, and the error of honest/trustworthy nodes. We wish to see how the algorithm behaves when the density of the nodes increases; we consider four cases with 500, 1 000, 1 500, and 2 000 nodes on the same area; thus we allow the density to increase by a factor of four. We also investigate the average trust when α, the ratio of malicious nodes to the total number of nodes increases from $\alpha = 0.2$ to $\alpha = 0.4$ and, finally, to $\alpha = 0.6$.

This straightforward data partitioning strategy for the distributed trust management algorithm is not a reasonable one for the centralized algorithm because it would lead to excessive communication among the cloud instances. Individual nodes may contribute data regarding primary transmitters in a different subarea; to evaluate the trust of each node, the cloud instances would have to exchange a fair amount of information. This data partitioning would also complicate our algorithm that groups together secondary nodes based on the distance from the primary one.

Instead, we allocate to each instance a number of channels, and all instances share the information about the geographic position of each node; the distance of a secondary node to any primary one can then be easily computed. This data partitioning strategy scales well in the number of primaries. Thus, it is suitable for simulation in large metropolitan areas but may not be able to accommodate cases when the number of secondaries is of the order of 10^8–10^9.

The objectives of our studies are to understand the limitations of the algorithm; the aim of the algorithm is to distinguish between trustworthy and malicious nodes. We expect that the ratio of malicious to trustworthy nodes, and the node density, should play an important role in this decision. The measures we examine are the average trust for all nodes and the average trust of individual nodes.

The effect of the malicious versus trustworthy node ratio on the average trust. We report the effect of the malicious versus trustworthy node ratio on the average trust when the number of nodes increases. The average trust is computed separately for the two classes of nodes and enables us to determine if the algorithm is able to clearly separate them.

Recall that the area is constant, thus, when the number of nodes increases, so does the node density. First, we consider two extreme case; the malicious nodes represent only 20% of the total number of nodes and an unrealistically high presence, of 60%. Then, we report on the average trust when the number of nodes is fixed and the malicious nodes represent an increasing fraction of the total number of nodes.

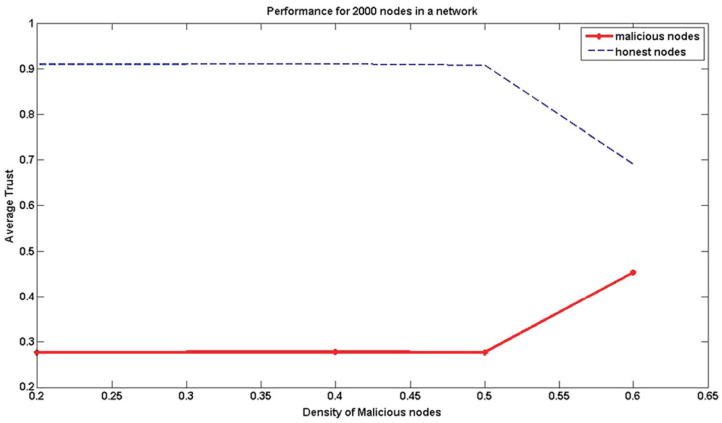


FIGURE A.4

The average trust function of α for a population size of 2 000 nodes. As long as malicious nodes represent 50% or less of the total number of nodes, the average trust of malicious nodes is below 0.3, while the one of trustworthy nodes is above 0.9 in a scale of 0 to 1.0. As the number of nodes increases, the distance between the average trust of the two classes becomes larger, and even larger when $\alpha > 0.5$, i.e., the malicious nodes are in majority.

Results reported in [63] show that, when the malicious nodes represent only 20% of all nodes, there is a clear distinction between the two groups. The malicious nodes have an average trust of 0.28, and the trustworthy ones have an average trust index of 0.91, regardless of the number of nodes.

When the malicious nodes represent 60% of all the nodes, then the number of nodes plays a significant role; when the number of nodes is small, the two groups cannot be distinguished since their average trust index is almost equal, 0.55, although the honest nodes have a slightly above-average trust value. When the number of nodes increases to 2 000 and the node density increases four folds, then the average trust of the first (malicious) group decreases to 0.45, and for the second (honest) group, it increases to about 0.68.

This result is not unexpected; it only shows that the history-based algorithm is able to classify the nodes properly even when the malicious nodes are a majority, a situation we do not expect to encounter in practice. This effect is somewhat surprising; we did not expect that, under these extreme conditions, the average of the trust of all nodes will be so different for the two groups. A possible explanation is that our strategy to reward constant good behavior, rather than occasional good behavior, and designed to mask the true intentions of a malicious node, works well.

Fig. A.4 shows the average trust function of α, the ratio of malicious versus total number of nodes. The results confirm the behavior discussed earlier; we see a clear separation of the two classes only when the malicious nodes are in the minority. When the density of malicious nodes approaches a high value so that they are in majority, the algorithm still performs as evident from the figure that the average trust for honest nodes even at high value of α is more than for malicious nodes. Thus, the trusts reflect the aim of isolating the malicious from the honest sets of nodes. We also observe that the separation is clearer when the number of nodes in the network increases.

The benefits of a cloud-based service for trust management, A cloud service for trust management in cognitive networks can have multiple technical and economical benefits [95]. The service is likely to have a broader impact than the one discussed here; it could be used to support a range of important policies in wireless network where many decisions require the cooperation of all nodes. A history-based algorithm to evaluate the trust and detect malicious nodes with high probability is at the center of the solution we have proposed [63].

A centralized, history-based algorithm for bandwidth management in CRNs has several advantages over the distributed algorithms discussed in the literature:

- It drastically reduces the computations a mobile device is required to carry out to identify free channels and avoid penalties associated with interference with primary transmitters.
- It enables a secondary node to get information about channel occupancy as soon as it joins the system and later on demand; this information is available even when a secondary node is unable to receive reports from its neighbors, or when it is isolated.
- It does not require the large number of assumptions critical to the distributed algorithms.
- The dishonest nodes can be detected with high probability, and their reports can be ignored; thus, over time, the accuracy of the results increases. Moreover, historic data could help detect a range of Byzantine attacks orchestrated by a group of malicious nodes.
- It is very likely to produce more accurate results than the distributed algorithm because the reports are based on information from all secondary nodes reporting on a communication channel used by a primary, not only those in its vicinity; a higher node density increases the accuracy of the predictions. The accuracy of the algorithm is a function of the frequency of the occupancy reports provided by the secondary nodes.

The centralized trust-management scheme has several other advantages. First, it can be used not only to identify malicious nodes and provide channel occupancy reports, but it also can manage the allocation of free channels. In the distributed case two nodes may attempt to use a free channel and collide; this situation is avoided in the centralized case. At the same time, malicious nodes can be identified with high probability and be denied access to the occupancy report.

The server could also collect historic data regarding the pattern of behavior of the primary nodes and use this information for the management of free channels. For example, when a secondary node requests access for a specific length of time, the service may attempt to identify a free channel likely to be available for that time.

The trust management may also be extended to other network operations, such as routing in a mobile ad hoc network; the strategy in this case would be to avoid routing through malicious nodes.

A.3 Simulation of traffic management in a smart city

The objective of the project is to study the traffic crossing the center of a city for variations in traffic intensities and traffic-light scheduling strategies.

The layout of the city center. Rectangular grid with n rows and m columns. There are NS (North–South) avenues and EW (East–West) streets. All avenues and streets are one way and have multiple lanes.

- The NS and SN avenues are $\mathcal{A}^{NS}(i)$ and $\mathcal{A}_i^{SN}$, $i \leq m$, respectively. The EW and WE streets are $\mathcal{S}_j^{EW}$ and S_j^{WE}, $j \leq n$, respectively.
- All distances are measured in c units; one unit equals the average car length plus an average required distance from the car ahead.
- The distance between rows i and $i+1$ and between columns j and $j+1$ are denoted by d_i, $1 \leq i \leq n$ and d_j, $1 \leq j \leq m$, respectively, and $\min(d_i, d_j) > kc$ with $k > 100$.
- The direction of one-way traffic alternates on both avenues and streets; $\mathcal{A}_j^{NS}$, $j \in \{1, 3, \ldots\}$ and $\mathcal{A}^{SN}$, $j \in \{2, 4, \ldots\}$; similarly, $\mathcal{S}_i^{EW}, i \in \{1, 3, \ldots\}$ and $\mathcal{S}^{WE}, i \in \{2, 4, \ldots\}$. Avenues and streets have either two or three lanes. Call L_j^{NS} the number of lanes of $\mathcal{A}_j^{NS}$.

The traffic lights are installed at all intersections $\mathcal{I}_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq m$. The traffic lights $\mathcal{I}_{i,j}$, $1 < i < n$, $1 < j < m$ allow left turns. Call $\tau_{i,j}^{gNS}(t)$, $\tau_{i,j}^{gNW}(t)$, $\tau_{i,j}^{gEW}(t)$ and $\tau_{i,j}^{gES}(t)$ the duration of the green light for the cycle starting at time t for directions NS, NW, EW, and ES, respectively, of traffic light $\mathcal{I}_{i,j}$.

Cars enter and exit the grid from all directions: NS, SN, EW, and WE.

- Each car has an associated path. For example, the path P^k of car C_i^k, $1 \leq i \leq MaxConv$ entering the grid is described by the pair entry point $\mathcal{I}_{i,j}^{k,entry}$ and exit point $\mathcal{I}_{i,j}^{k,exit}$, with $i = 1$ or $i = n$, and $j = 1$ or $j = m$, and at most two intermediate turning points $\mathcal{I}_{i,j}^{k,turn1}$ and $\mathcal{I}_{i,j}^{k,turn2}$ with $1 < i < n$ and $1 < j < m$. Call t_i^{in} the time when a car enters the grid.
- Cars entering the grid are grouped in "convoys" of various sizes. Convoys can be split when cars leave it to turn left or right or when the light turns red. Convoys are merged when cars enter the convoy or when cars from another convoy join one convoy stopped at a traffic light.
- Cars whose paths requires a right turn should be in the right lane of an avenue or street, and cars whose path requires a left turn should be in the left lane. Cars that do not turn should be in the center lane.
- Call $v_{i,j}^{in}(t, s)$ and $v_{p,q}^{out}(t, s)$ the number of cars in the convoys entering and, respectively, exiting the grid at intersections $\mathcal{I}_{i,j}$ and $\mathcal{I}_{p,q}$, in the interval $s - t$.
- Call $\Phi_{i,j}^{in}(t, s)$ the traffic intensity as the number of cars entering the grid in the interval $s - t$.

Hints for project implementation. Create a description file with separate sections describing: (i) the layout including n, m, c; for example, n = 100, m = 80, and c = 12 m; (ii) the initial traffic lights setting; and (iii) the car arrival process and the convoys at each entry point. The simulation will require several data structures including:

- Car records. Each record includes: (i) static data such as: the CarId, the entry point, the exit point, the entry time, and the path; and (ii) dynamic data such as: a list of all traffic lights it had to wait for the green light and the waiting time, the speed on each block, and the time of exit.
- Traffic-light records. Each record should include:

  1. static data such as: TrafficLightId; the location as the intersection of avenue $\mathcal{A}_i^{XX}$ with street $\mathcal{S}_j^{YY}$;
  2. dynamic data such as: the number of cycles (red + yellow + green); for each cycle, it should include the times for green in each direction and the identity of convoys waiting.

To evaluate the performance, compute the *average transition time* of all cars in the interval (s, t) for several traffic management algorithms:

Dumb scheduling—traffic lights follow a static, deterministic schedule.
Individual self-managed scheduling—a traffic light switches to green in direction DD when the length of the queue of cars waiting to cross the intersection exceeds a threshold L_{sw}.
Coordinated scheduling—modify the previous algorithm to include communication among neighboring nodes; use a publish–subscribe algorithm in which each traffic light subscribes to messages sent by its four neighbors.
Convoy-aware algorithm—attempts to create a green wave for the largest convoys and anticipate the setting of green lights to next intersections at the time the convoy reaches the intersection.

Project implementation.[3] The simulation includes a graphical user interface displaying the smart city center. The four algorithms are called: Dumb, Self-Managed, Coordinated, and Convoy. The implementation uses Java Swing API and defines several classes:

1. Road—manages the layout of the grid.
2. Traffic Point—handles intersections and entry and exit points.
3. PaintGrid—is responsible for continuously painting and repainting the grid for a given time interval.
4. Frame—sets the size of the Frame Window used by the Canvas to draw the grid.
5. Car—creates cars and feeds them to a Java hashmap dynamic list.
6. Schedule—manages changing of the traffic lights, and implements the logic of the four scheduling algorithms and controls the car arrival process.
7. Convoy—generates new convoy and adds cars to a convoy.
8. Statistics—gathers traffic statistics.
9. StatWindow—displays traffic statistics.

The car arrival stochastic process has an exponential distribution of the interarrival times

$$p(t) = \lambda e^{-\lambda t}. \tag{A.18}$$

Lower values of parameter λ in Eq. (A.18) result in higher concentration of cars for the first few simulation cycles, while higher values of *lambda* scatter cars on larger number of simulation cycles; see Fig. A.5. A record describing the path of a car is created at the time when a car enters the grid. The path of a car could have zero turns if the car enters and exit on the same street or avenue. The path will have one turn if the car enters on one street and exits from an avenue, or enters an avenue and exits a street. The path will have two turns if the car enters one street and exits from another street or enters an avenue and exits from a different avenue.

The next step involves the logic of car movement through an intersection. The traffic light checks if the car is moving straight or turning, using car path information. The state of the traffic-light status determines whether the car should decelerate to stop or move ahead and accelerate towards the next intersection. If the car turns, then a car speed-switch function is called.

[3] The implementation of the project is due to the group of five students: Ahmed Alhazmi, Austin Jerome, Ahmad Qutbuddin, Sai Lalitha Renduchintala, and Wendelyn Sanabria.

Shown next is a segment of the configuration file specifying the layout and some of parameters related to cars.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class Configuration {
        // Simulation class:
        //      Exponential Car Insertion Rate; Number of Cars
        protected int Lambda = 15;
        protected int NumberOfCars = 350;
        // Grid class:
        //      Number of Streets and Avenues
        //      Maximum and Minimum Block Side Length in c units
        protected int NumberOfStreets = 3;
        protected int NumberOfAvenues = 3;
        protected int MaximumBlockSide = 35;
        protected int MinimumBlockSide = 35;
        // Road class:
        //      Number of Forward and Turning Lanes
        protected int NumberOfForwardLanes = 2;
        protected int NumberOfTurningLanes = 1;
        // TrafficLight class
        //      Maximum Red and Green time in seconds
        //      Maximum Red Time could be Maximum Green Time + Yellow Time
        //      Yellow Time in seconds; Intersection light initial status (TBD)
        //      Scheduling Scheme D, S, C, V
        protected int MaxRedTime = 4000;
        protected int MaxGreenTime = 3000;
        protected int YellowTime = 1000;
        protected char SchedulingScheme = 'C';
        // Car class:
        //      Maximum Car Speed in c/second unit
        //      Car Acceleration in c/second2 unit; Car length and width in pixels
        protected int CarSpeed = 5;
        protected int CarAcceleration = 1;
        protected int CarLength = 6;
        protected int CarWidth = 3;
        protected int Clearance = 2;
```

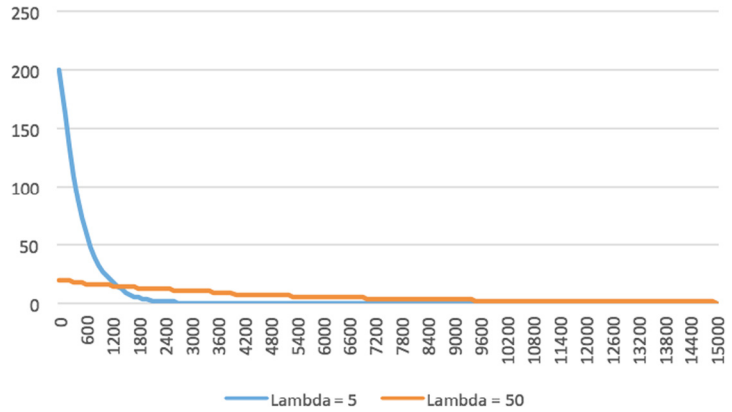


FIGURE A.5

The number of cars entering the system function of time represented by simulation cycles. Higher values of the parameter λ in Eq. (A.18) scatter cars for a larger number of simulation cycles.

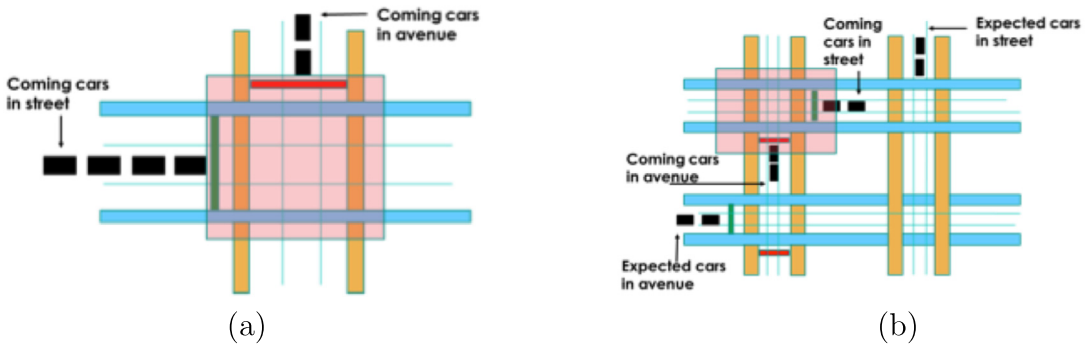


FIGURE A.6

(a) Self-managed scheduling uses the information about the number of cars to change the traffic light. (b) In the coordinated scheduling, self-managed intersections coordinate their traffic lights.

The self-managed scheduling uses two queues for the traffic lights at the intersection of streets and avenues. The length of the car queue at the traffic lights triggers changes of the traffic lights, as seen in Fig. A.6(a).

The coordinated scheduling algorithm was implemented as an extension of the self-managed scheduling algorithm. Now, an intersection communicates with its neighboring intersections to determine the number of cars that are passing through. This allows the next intersection along the path to turn green when a larger number of cars will reach the intersection; see Fig. A.6(b).

The convoy scheduling algorithm treats a number of n cars separated by a car length, c, to be assimilated with the larger car of length $n \times c$. Once a convoy is in motion, a traffic light extends its

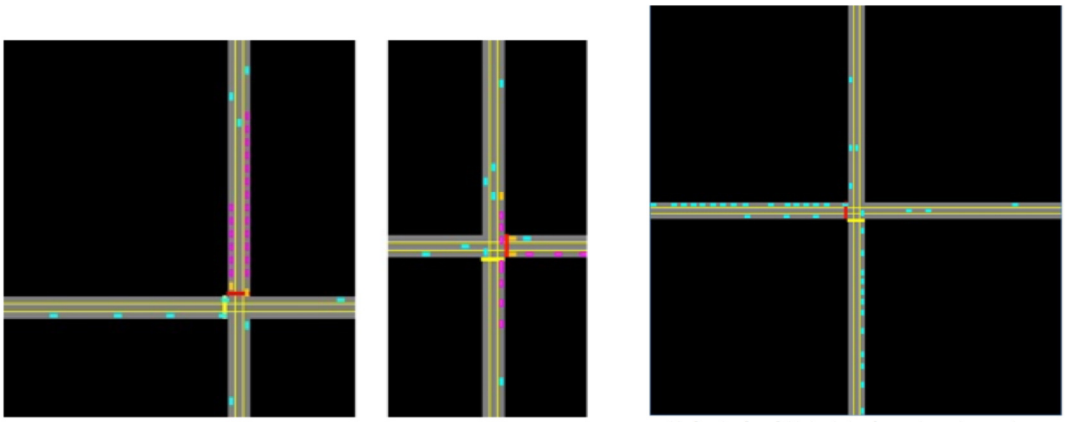


FIGURE A.7

(a) Convoy of cars waiting at an intersection (left); traffic lights extend the yellow-light holds to allow a convoy to cross the intersection (right). (b) The simulation of maximum capacity scenario based on an ideal orchestration.

yellow state to allow the entire convoy to pass. Once a car turns, the whole convoy is broken up; see Fig. A.7(a).

We have also investigated an ideal orchestrations scenario when the distance between cars allows the two continuous streams of cars entering every intersection from the two directions to pass alternatively through without stopping. This ideal traffic light-less scenario seen in Fig. A.7(b) allows us to determine the maximum capacity of the grid.

Some of the simulation results are discussed next. The average transit time decreases steadily from about 36.8 units of simulated time for the dumb traffic lights scheduling algorithm to 32.6 for the self-managed, 32 for the coordinated, and 31.4 for the convoy. The average waiting time is 19.8, 17.4, and 15.2 for the self-managed, coordinated, and convoys traffic lights scheduling algorithms, respectively.

Fig. A.8 shows also a comparison of four algorithms. For each algorithm, it shows the waiting time for entering the grid, the transit time through the grid, and the waiting time at traffic lights while transiting through grid. The convoy scheduling algorithm performs best. Fig. A.9 shows the confidence intervals from 50 simulation runs for the coordinated-scheduling algorithm.

A.4 A cloud service for adaptive data streaming

In this section, we discuss a cloud application related to data streaming [394]. Data streaming is the name given to the transfer of data at a high rate with real-time constraints. Multimedia applications, such as music and video streaming, high-definition television (HDTV), scientific applications that process a continuous stream of data collected by sensors, the continuous backup copying to a storage medium of the data flow within a computer, and many other applications require the transfer of real-time data

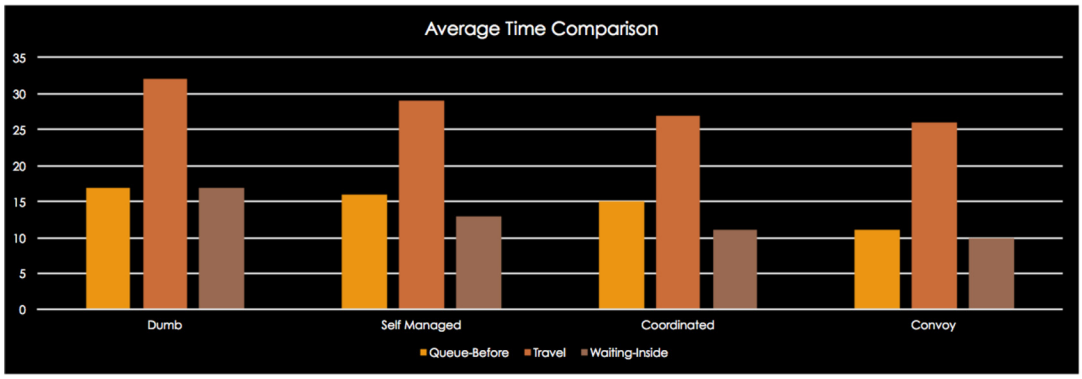


FIGURE A.8

The waiting time before entering the grid, the transit time through the grid, and the waiting time at traffic lights while transiting the grid. The results displayed are for the four traffic-lights scheduling algorithms, dumb, self-managed, correlated, and convoys.

	Total Number of Cars	Average Transient Time	Average Waiting Time
Maximum	495	36	23
Minimum	354	31	13
Average	431.32	32.92	17.94
Standard Deviation	34.83299585	1.11067547	2.395078287
95 % Confidence	9.655036433	0.307857876	0.663869631
Upper	441	33	19
Lower	421	32	17

FIGURE A.9

Confidence intervals for the car transit time.

at a high rate. For example, to support real-time human perception of the data, multimedia applications have to make sure that enough data is being continuously received without any noticeable time lag.

We are concerned with the case when the data streaming involves a multimedia application connected to a service running on a computer cloud. The stream could originate from the cloud, as is the case of the iCloud service provided by Apple, or could be directed toward the cloud, as in the case of a real-time data collection and analysis system.

Data streaming involves three entities: the sender, a communication network, and a receiver. The resources necessary to guarantee the timing constraints include CPU cycles and buffer space at the sender and the receiver and network bandwidth. Adaptive data streaming determines the data rate based on the available resources. Lower data rates imply lower quality, but they reduce the demands for system resources.

Adaptive data streaming is possible only if the application permits trade-offs between quantity and quality. Such trade-offs are feasible for audio and video streaming that allow lossy compression, but are not acceptable for many applications that process a continuous stream of data collected by sensors.

Data streaming requires accurate information about all resources involved, and this implies that the network bandwidth has to be constantly monitored; at the same time, the scheduling algorithms should be coordinated with memory management to guarantee the timing constraints. Adaptive data streaming poses additional constraints because the data flow is dynamic. Indeed, once we detect that the network cannot accommodate the data rate required by an audio or video stream we have to reduce the data rate to convert to a lower-quality audio or video. Data conversion can be done on the fly, and, in this case, the data flow on the cloud has to be changed.

Accommodating dynamic data flows with timing constraints is nontrivial; only about 18% of the top 100 global video web sites use ABR (Adaptive Bit Rate) technologies for streaming [460].

This application stores the music files in S3 buckets, and the audio service runs on the EC2 platform. In EC2, each virtual machine functions as a virtual private server and is called an *instance;* an instance specifies the maximum amount of resources available to an application and the interface for that instance, as well as the cost per hour.

EC2 allows the import of VM images from the user environment to an instance through a facility called *VM import.* It also distributes automatically the incoming application traffic among multiple instances using the *elastic load balancing* facility. EC2 associates an *elastic IP address* with an account. This mechanism allows a user to mask the failure of an instance and remap a public IP address to any instance of the account, without the need to interact with the software support team.

The adaptive audio streaming involves a multiobjective optimization problem. We wish to convert the highest quality audio file stored on the cloud to a resolution corresponding to the rate that can be sustained by the available bandwidth; at the same time, we wish to minimize the cost on the cloud site and also minimize the buffer requirements for the mobile device to accommodate the transmission jitter. Finally, we wish to reduce to a minimum the start-up time for the content delivery.

A first design decision is whether or not data streaming should only begin after the conversion from the WAV to MP3 format has been completed, or if it should proceed concurrently with conversion, in other words, start as soon as several MP3 frames have been generated; another question is whether or not the converted music file should be saved for later use or discarded.

To answer these questions, we experimented with conversion from the highest quality audio files, which require a 320 Kbps data rate to lower quality files corresponding to 192, 128, 64, 32, and finally 16 Kbps. If the conversion time is small and constant, there is no justification for pipelining data conversion and streaming, a strategy that complicates the processing flow on the cloud. It makes sense to cache the converted copy for a limited period of time with the hope that it will be reused in the next future.

Another design decision is how the two services should interact to optimize the performance; two alternatives come to mind:

1. The audio service running on the EC2 platform requests the data file from the S3, converts it, and, eventually, sends it back. The solution involves multiple delays and it is far from optimal.
2. The S3 bucket is mounted as an EC2 drive. This solution reduces considerably the start-up time for audio streaming.

Table A.1 Conversion time in s on a *EC2 t1.micro* server platform; the source file is of the highest audio quality, 320 Kbps. The individual conversions are labeled $C1$ to $C10$; $\bar{T}_c$ is the mean conversion time.

Bit-rate (Kbps)	Audio file size (MB)	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\bar{T}_c$
192	6.701974	73	43	19	13	80	42	33	62	66	36	46.7
128	4.467982	42	46	64	48	19	52	52	48	48	13	43.2
64	2.234304	9	9	9	9	10	26	43	9	10	10	14.4
32	1.117152	7	6	14	6	6	7	7	6	6	6	7.1
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

Table A.2 Conversion time in s on a *EC2 t1.micro* server platform; the source file is of high audio quality, 192 Kbps. The individual conversions are labeled $C1$ to $C10$; $\bar{T}_c$ is the mean conversion time.

Bit-rate (Kbps)	Audio file size (MB)	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\bar{T}_c$
128	4.467982	14	15	13	13	73	75	56	59	72	14	40.4
64	2.234304	9	9	9	32	44	9	23	9	45	10	19.9
32	1.117152	6	6	6	6	6	6	20	6	6	6	7.4
16	0.558720	6	6	6	6	6	6	20	6	6	6	5.1

The conversion from a high-quality audio file to a lower quality, thus a lower bit rate, is performed using the LAME library.

The conversion time depends on the desired bit-rate and the size of the original file. Tables A.1, A.2, A.3, and A.4 show the conversion time in s when the source MP3 file are of 320 Kbps and 192 Kbps, respectively; the size of the input files is also shown.

The platforms used for conversion are: (a) the *EC2 t1.micro* server for the measurements reported in Tables A.1 and A.2 and (b) the *EC2 c1.medium* for the measurements reported in Tables A.3 and A.4. The instances run the Ubuntu Linux operating system.

The results of our measurements when the instance is the *t1.micro* server exhibit a wide range of the conversion times, 13–80 s, for the large audio file of about 6.7 MB when we convert from 320 to 192 Kbps. A wide range, 13–64 s, is also observed for an audio file of about 4.5 MB when we convert from 320 to 128 Kbps. For low-quality audio, the file size is considerably smaller, about 0.56 MB, and the conversion time is constant and small, 4 s.

Fig. A.10 shows the average conversion time for the experiments summarized in Tables A.1 and A.2. It is somewhat surprising that the average conversion time is larger when the source file is smaller as it is in the case when the target bit rates are 64, 32 and 16 Kbps. Fig. A.11 shows the average conversion time for the experiments summarized in Tables A.3 and A.4.

The results of our measurements when the instance runs on the EC2 c1.medium platform show consistent and considerably lower conversion times; Fig. A.11 presents the average conversion time.

To understand the reasons for our results, we took a closer look at the two types of EC2 instances, "micro" and "medium," and their suitability for the adaptive data-streaming service. The t1.micro sup-

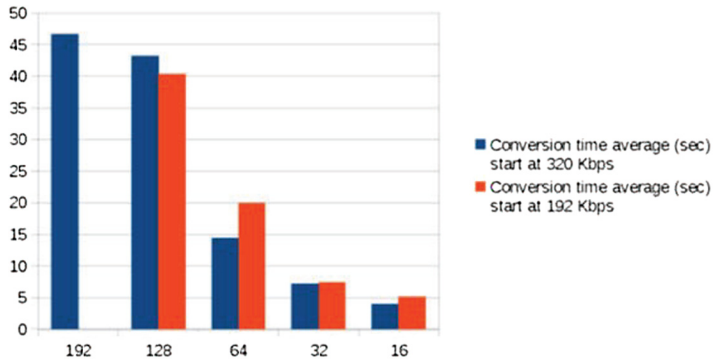


FIGURE A.10

The average conversion time on a *EC2 t1.micro* platform. The left bars and the right bars correspond to the original file at: the highest resolution (320 Kbps data rate) and next highest resolution (192 Kbps data rate), respectively.

Table A.3 Conversion time T_c in seconds on a *EC2 c1.medium* platform; the source file is of the highest audio quality, 320 Kbps. The individual conversions are labeled $C1$ to $C10$; $\bar{T}_c$ is the mean conversion time.

Bit-rate (Kbps)	Audio file size (MB)	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\bar{T}_c$
192	6.701974	15	15	15	15	15	15	15	15	15	15	15
128	4.467982	15	15	15	15	15	15	15	15	15	15	15
64	2.234304	11	11	11	11	11	11	11	11	11	11	11
32	1.117152	7	7	7	7	7	7	7	7	7	7	7
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

Table A.4 Conversion time in s on a *EC2 c1.medium* platform; the source file is of high audio quality, 192 Kbps). The individual conversions are labeled $C1$ to $C10$; $\bar{T}_c$ is the mean conversion time.

Bit-rate (Kbps)	Audio file size (MB)	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	$\bar{T}_c$
128	4.467982	15	15	15	15	15	15	15	15	15	15	15
64	2.234304	10	10	10	10	10	10	10	10	10	10	10
32	1.117152	7	7	7	7	7	7	7	7	7	7	7
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

ports bursty applications, with a high average-to-peak ratio for CPU cycles, e.g., transaction processing systems. EBS provides block level storage volumes; the "micro" instances are only EBS-backed.

The "medium" instances support compute-intensive applications with a steady and relatively high demand for CPU cycles. Our application is compute-intensive, thus there should be no surprise that our measurements for the EC2 c1.medium platform show consistent and considerably lower conversion times.

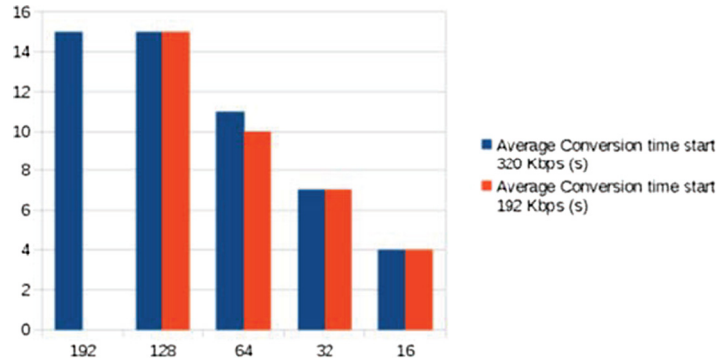


FIGURE A.11

The average conversion time on a *EC2 c1.medium* platform. The left bars and the right bars correspond to the original file at: the highest resolution (320 Kbps data rate) and the next highest resolution (192 Kbps data rate), respectively.

A.5 Optimal FPGA synthesis

We now discuss another class of applications that could benefit from cloud computing. The benchmarks presented in Section 11.10 compared the performance of several codes running on a cloud with runs on supercomputers. As expected, the results showed that a cloud is not an optimal environment for applications exhibiting fine- or medium-grained parallelism. Indeed, communication latency is considerably larger on a cloud than on a supercomputer with a more expensive custom interconnect. This simply means that we have to identify applications that do not involve extensive communication, or applications exhibiting coarse-grained parallelism.

Computer clouds provide an ideal running environment for scientific applications involving model development when multiple cloud instances could concurrently run slightly different models of the system. When the model is described by a set of parameters, the application can be based on the SPMD paradigm combined with an analysis phase when the results from the multiple instances are ranked based on a well-defined metric.

In this case, there is no communication during the first phase of the application when partial results are produced and then written to storage server. The individual instances signal the completion, and a new instance to carry out the analysis and display the results is started. A similar strategy can be used by engineering applications of mechanical, civil, electrical, or electronic engineering, or any other system design area. In this case, the multiple instances run a concurrent design for different sets of parameters of the system.

A cloud application for optimal design of field-programmable gate arrays (FPGAs) is discussed next. As the name suggests, an FPGA is an integrated circuit designed to be configured/adapted/programmed in the field to perform a well-defined function [428]. Such a circuit consists of *logic blocks* and *interconnects* that can be "programmed" to carry out logical and/or combinatorial functions; see Fig. A.12.

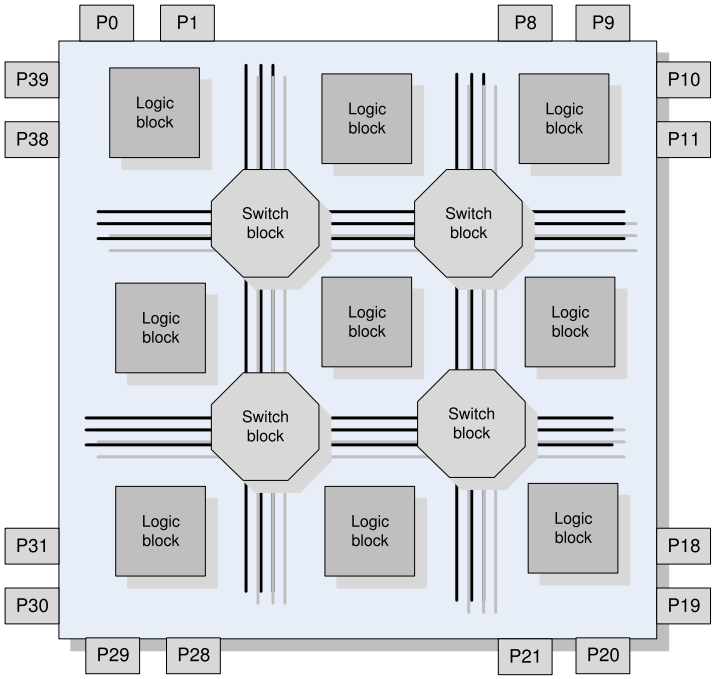


FIGURE A.12

The structure of a Field Programmable Gate Array (FPGA) with 30 pins, $P1$-$P29$, 9 logic blocks, and 4 switch-blocks.

The first commercially viable FPGA, XC2064, was produced in 1985 by Xilinx. Today, FPGAs are used in many areas, including digital signal processing, CRNs, aerospace, medical imaging, computer vision, speech recognition, cryptography, and computer hardware emulation. FPGAs are less energy efficient and slower than application-specific integrated circuits (ASICs). The widespread use of FPGAs is due to their flexibility and the ability to reprogram them.

Hardware description languages (HDLs) such as VHDL and Verilog are used to program FPGAs; HDLs are used to specify a register-transfer level (RTL) description of the circuit. Multiple stages are used to synthesize FPGA.

A cloud-based system was designed to optimize the routing and placement of components. The basic structure of the tool is shown in Fig. A.13. The system uses the PlanAhead tool from Xilinx—see http://www.xilinx.com/—to place system components and route chips on the FPGA logical fabric. The computations involved are fairly complex and take a considerable amount of time; for example, a fairly simple system consisting of a software core processor (Microblaze), a block random access memory (BRAM), and a couple of peripherals can take up to 40 min to synthesize on a powerful workstation. Running N design options in parallel on a cloud speeds up the optimization process by a factor close to N.

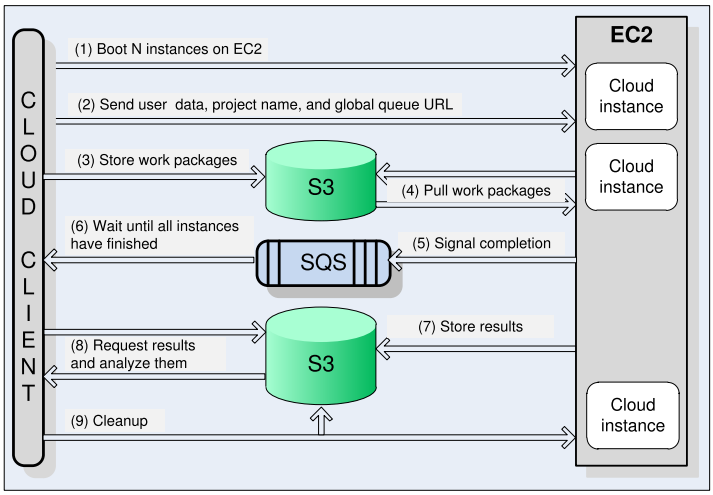


FIGURE A.13

The architecture of a cloud-based system to optimize the routing and placement of components on an FPGA.

A.6 Tensor network contraction on AWS

A numerical simulation project related to research in condensed matter physics is discussed in [332] and overviewed in this section. To illustrate the problems posed by Big Data, we analyze various options offered by 2016-vintage Amazon Web Services for running the application. *M4* and *C4* seem to be the best choices for applications such as Tensor Network Contraction (TNC).

Tensor contraction. In linear algebra, the *rank* $\mathcal{R}$ of an object is given by the number of indices necessary to describe its elements. A scalar has rank 0, a vector $a = (a_1, a_2, \ldots, a_n)$ has rank 1 and n elements, and a matrix $\mathcal{A} = [a_{ij}]$, $1 \leq i \leq n$, $1 \leq j \leq m$ has rank 2 and $n \times m$ elements

$$\mathcal{A} = \begin{bmatrix} a_{11} & a_{12} & \ldots a_{1m} \\ a_{21} & a_{22} & \ldots a_{2m} \\ \vdots & & \\ a_{n1} & a_{n2} & \ldots a_{nm} \end{bmatrix}. \tag{A.19}$$

Tensors have rank $\mathcal{R} \geq 3$; the description of tensor elements is harder. For example, consider a rank 3 tensor $\mathcal{B} = [b_{jkl}]$ with elements $b_{jkl} \in \mathbb{R}^{2\times 2\times 2}$. The eight elements of this tensor are: $\{b_{111}, b_{112}, b_{121}, b_{122}\}$ and $\{b_{211}, b_{212}, b_{221}, b_{222}\}$. We can visualize the tensor elements as the vertices of a cube where the first group of elements are in the plane $j = 1$ and $j = 2$, respectively. Similarly, the tensor elements $\{b_{111}, b_{211}, b_{112}, b_{212}\}$ and $\{b_{121}, b_{221}, b_{122}, b_{222}\}$ are in the planes $k = 1$ and $k = 2$, respectively, while $\{b_{111}, b_{121}, b_{211}, b_{221}\}$ and $\{b_{112}, b_{122}, b_{212}, b_{222}\}$ are in the planes $l = 1$ and $l = 2$, respectively.

Tensor contraction is the summation over repeated indices of the two tensors or of a vector and a tensor. Let $\mathcal{C}$ be the contraction of two arbitrary tensors $\mathcal{A}$ and $\mathcal{B}$. The rank of the tensor resulting after contraction is

$$\mathcal{R}(\mathcal{C}) = \mathcal{R}(\mathcal{A}) + \mathcal{R}(\mathcal{B}) - 2. \tag{A.20}$$

For example, when $\mathcal{A} = [a_{ij}]$, $\mathcal{B} = [b_{jkl}]$ and we contract over j, we obtain $\mathcal{C} = [c_{ikl}]$ with

$$c_{ikl} = \sum_j a_{ij} b_{jkl}. \tag{A.21}$$

The rank of $\mathcal{C}$ is $\mathcal{R}(\mathcal{C}) = 2 + 3 - 2 = 3$. Tensor $\mathcal{C}$ has eight elements:

$$\begin{array}{l|l}
c_{111} = \sum_{j=1}^{2} a_{1j} b_{j11} = a_{11} b_{111} + a_{12} b_{211} & c_{121} = \sum_{j=1}^{2} a_{1j} b_{j21} = a_{11} b_{121} + a_{12} b_{221} \\
c_{212} = \sum_{j=1}^{2} a_{2j} b_{j12} = a_{21} b_{112} + a_{22} b_{212} & c_{222} = \sum_{j=1}^{2} a_{2j} b_{j22} = a_{21} b_{122} + a_{22} b_{222} \\
c_{112} = \sum_{j=1}^{2} a_{1j} b_{j11} = a_{11} b_{112} + a_{12} b_{212} & c_{122} = \sum_{j=1}^{2} a_{1j} b_{j21} = a_{11} b_{122} + a_{12} b_{222} \\
c_{211} = \sum_{j=1}^{2} a_{2j} b_{j11} = a_{21} b_{111} + a_{22} b_{211} & c_{221} = \sum_{j=1}^{2} a_{2j} b_{j21} = a_{21} b_{121} + a_{22} b_{221}
\end{array} \tag{A.22}$$

Tensor networks and tensor network contraction (TNC). A *tensor network* is defined as follows: Let $[A_1], \ldots, [A_n]$ be n tensors with index sets $x^{(1)}, \ldots, x^{(n)}$ where each $\{x^{(i)}\}$ is a subset of $\{x_1, \ldots, x_N\}$ with N very large. We assume that the "big" tensor $[A]_{\{x_1,\ldots,x_N\}}$ can be expressed as the product of the "smaller" tensors $[A_1], \ldots, [A_n]$:

$$[A]_{\{x_1,\ldots,x_K\}} = [A_1]_{\{x^{(1)}\}} \ldots [A_n]_{\{x^{(n)}\}}. \tag{A.23}$$

We wish to compute the scalar

$$Z_A = \sum_{\{x_1,\ldots,x_N\}} [A_i]_{\{x_1,\ldots,x_N\}}. \tag{A.24}$$

For example, $N = 7$ and $n = 4$ and the index set is $\{x_1, x_2, \ldots, x_7\}$ for the TNC in Fig. A.14. The four "small" tensors and their respective subsets of the index set are

$$[A_1]_{\{x_1,x_2,x_5\}},\ [A_2]_{\{x_2,x_3,x_4\}},\ [A_3]_{\{x_3,x_4,x_6\}} \text{ and } [A_4]_{\{x_5,x_6,x_7\}}. \tag{A.25}$$

The "big" tensor $[A]$ is the product of the four "small" tensors

$$[A]_{\{x_1,x_2,x_3,x_4,x_5,x_6,x_7\}} = [A_1]_{\{x_1,x_2,x_5\}} \otimes [A_2]_{\{x_2,x_3,x_4\}} \otimes [A_3]_{\{x_3,x_4,x_6\}} \otimes [A_4]_{\{x_5,x_6,x_7\}}. \tag{A.26}$$

To calculate Z_A, we first contract $[A_2]$ and $[A_3]$, and the result is tensor $[B]$

$$[B]_{\{x_2,x_6\}} = \sum_{x_3} \sum_{x_4} [A_2]_{\{x_2,x_3,x_4\}} \otimes [A_3]_{\{x_3,x_4,x_6\}}. \tag{A.27}$$

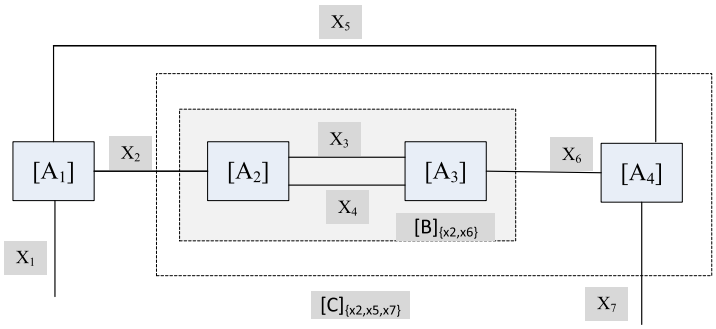


FIGURE A.14

The ordering of tensor contraction when the index set is $\{x_1, x_2, \ldots, x_7\}$ and the four "small" tensors are $[A_1]_{\{x_1,x_2,x_5\}}$, $[A_2]_{\{x_2,x_3,x_4\}}$, $[A_3]_{\{x_3,x_4,x_6\}}$ and $[A_4]_{\{x_5,x_6,x_7\}}$. Tensors $[B]$ and $[C]$ are the results of contraction of $[A_2]$, $[A_3]$, and $[A_4]$, $[B]$, respectively.

Next, we contract $[B]$ and $[A_4]$ to produce $[C]$

$$[C]_{\{x_2,x_5,x_7\}} = \sum_{x_6} [B]_{\{x_2,x_6\}} \otimes [A_4]_{\{x_5,x_6,x_7\}}. \tag{A.28}$$

Finally, we compute

$$Z_{\{x_1,x_7\}} = \sum_{x_2} \sum_{x_5} [A_1]_{\{x_1,x_2,x_5\}} \otimes [C]_{\{x_2,x_5,x_7\}}. \tag{A.29}$$

Tensor network contraction is CPU- and memory-intensive. If the tensor network has an arbitrary topology, TNC is considerably more intensive than in the case of a regular topology, e.g., a 2-D lattice.

An example of TNC. We now discuss the case of an application where the tensors form a 2-D, $L \times L$ rectangular lattice. Each tensor in the interior of the lattice has four indices, each one running from 1 to D^2, while outer tensors have only three indices, and the ones at the corners have only two. The resulting tensors form a product of vectors (top and bottom tensors) and matrices (interior tensors) with vertical orbitals running from 1 to D^{2L}. The space required for tensor network contraction can be very large: We expect parameter values as large as $D = 20$ and $L = 100$. This is a Big Data application; 20^{200} is a very large number indeed!!

Fig. A.15 illustrates the *generic TLC algorithm* for $L = 8$. The first iteration of the computation contracts the left-most (1L) and the right-most columns (1R) of the tensor network. The process continues until we end up with the "big" vector after $L/2 = 4$ iterations. The left and right contractions, (1L, 2L, and 3L) and (1R, 2R, and 3R), are mirror images of one another and are carried out concurrently.

TNC algorithm for condensed matter physics. In quantum mechanics, vectors in an n-dimensional Hilbert space describe quantum states, and tensors describe transformations of quantum states. Tensor network contraction has applications in condensed matter physics, and our discussion is focused only on the algorithmic aspects of the problem.

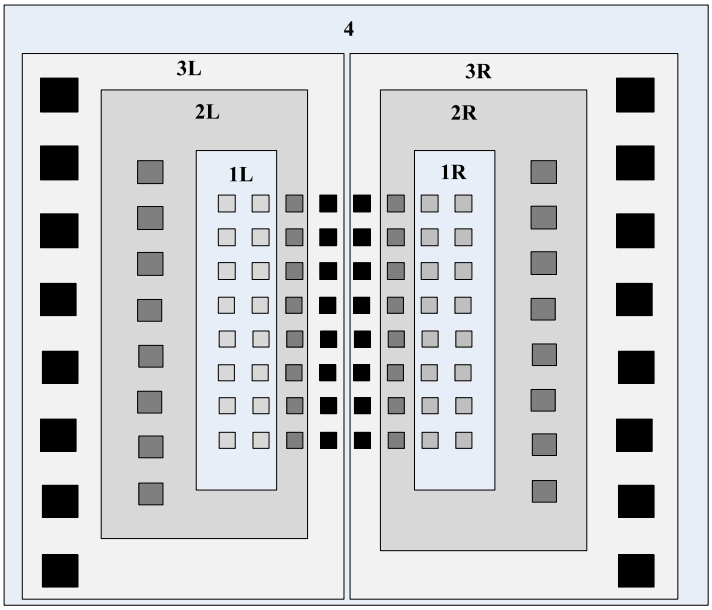


FIGURE A.15

Contraction when $L = 8$ and we have an 8×8 tensor network. The first iteration contracts columns 1 and 2 and columns 7 and 8; see 1L and 1R boxes. During the second iteration, the two resulting tensors are contracted with columns 3 and 6, respectively, as shown by 2L and 2R boxes. During the 3rd iteration, the new tensors are contracted with columns 4 and 5, respectively, as shown by the 3L and 3R boxes. Finally, during the 4th iteration, the "big" vector is obtained by contracting the two tensors produced by the 3rd iteration.

The algorithm for tensor network contraction should be flexible, efficient, and cost effective. Flexibility means the ability to run problems of various sizes, with a range of values for D and L parameters. An efficient algorithm should support effective multithreading and optimal use of available system resources.

The notations used to describe the contraction algorithms for tensor network T are:

- $N^{(i)}$—number of vCPUs for iteration i; $N = 2$ for all iterations of Stage 1, while the number of vCPUs for Stage 2 may increase with the number of iterations;
- m—the amount of memory available on the vCPU of the current instance;
- $\mathcal{T}^{(i)} = [\mathcal{T}^{(i)}_{j,k}]$—version of the $[T]$ after the ith iteration;
- $L^{(i)}$—the number of columns of $[T]$ at iteration i;
- $\mathcal{T}^{(i)}_{j,k}$—tensor in row j and column k of $\mathcal{T}^{(i)}$; $T^{1}_{j,k} = T_{j,k}$;
- $\mathcal{T}^{(i)}_{k}$—column k of $\mathcal{T}^{(i)}$;
- $\mathbb{C}(\mathcal{T}^{(i)}_{k}, T_j)$, $i > 1$—contraction operator applied to columns $\mathcal{T}^{(i)}_{k}$ and T_j in Stage 1;
- $\mathbb{V}(\mathcal{T}^{(L_{col})})$—vertical contraction operator applied to the "big tensor" obtained after L_{col} column contractions;

- μ—amount of memory for $T_{j,k}$, a tensor of the original T;
- $\mu^{(i)}$—storage for a tensor $\mathcal{T}_{j,k}^{(i)}$ created at iteration i;
- I_{max}—maximum number of iterations for Stage 1 of the TNC algorithm.

The *generic contraction algorithm* for a 2-D tensor network, with L_{row} rows and L_{col} columns, $T = [T_{j,k}]$, $1 \leq j \leq 2L_{row}$ $1 \leq k \leq 2L_{col}$, is an extension of the one in Fig. A.15. TNC is an iterative process: At each iteration, two pairs of columns are contracted concurrently. During the first iteration the two pairs of columns of T, $(1, 2)$, and $(2L, 2L-1)$ are contracted. At iterations $2 \leq i \leq L$, the new tensor network has $L^{(i)} = L - 2i$ columns, and the contraction is applied to column pairs: the column resulting from the contractions at iteration $(i-1)$, now columns 1 and $L^{(i)}$ with columns 2 and $L^{(i)} - 1$, respectively.

The TNC algorithm is organized in multiple stages with different AWS instances for different stages. Small- to medium-sized problems need only the first stage to produce the results and use low end instances with 2 vCPUs, while large problems must feed the results of the first stage to a second one running on more powerful AWS instances. A third stage may be required for extreme cases when the size of one tensor exceeds the amount of vCPU memory, some 4 GB at this time. The three stages of the algorithm are discussed next.

- *Stage 1.* An entire column of $\mathcal{T}^{(i)}$ can be stored in the vCPU memory and successive contraction iterations can proceed seamlessly when

$$L_{row}\left(2\mu + \mu^{(i-1)} + \mu^{(i)}\right) < m. \tag{A.30}$$

  This is feasible for the first iterations of the algorithm and for relatively small values of L_{row}. Call I_{max} the largest value of i which satisfies the Eq. (A.30).
  Use a low-end M4 or C4 instance with 2 vCPUs, $N = 2$. The computation runs very fast with optimal use of the secondary storage and network bandwidth. Each vCPU is multithreaded: Multiple threads carry out the operations required by the contraction operator $\mathbb{C}$, while one thread reads the next column of the original tensor network, $\mathcal{T}$ in preparation for the next iteration.
- *Stage 2.* After a number of iterations, the condition in Eq. (A.30) is no longer satisfied, and the second phase should start. Now an iteration consists of partial contractions when subsets of column tensors are contracted independently. In this case the number of vCPUs is $N > 2$.
- *Stage 3.* As the amount of space needed for a single tensor increases and the vCPU memory cannot store a single tensor,

$$\mu_i > m. \tag{A.31}$$

  In this extreme case we use several instances with the largest number of vCPUs, e.g., either M4.10xlarge or C4.10xlarge.

Stage 1 TNC algorithm. The algorithm is a straightforward implementation of the generic TNC algorithm:

1. Start an instance with $N = 2$, e.g., C4.large;
2. Read input parameters e.g., L_{row}, L_{col};
3. Compute I_{max};

4. First iteration
   a. vCP1—read T_1 and T_2, apply $\mathbb{C}(T_1, T_2)$; start reading T_3;
   b. vCP2—read $T_{L_{col}}$ and $T_{L_{col}-1}$, apply $\mathbb{C}(T_{L_{col}}, T_{L_{col}-1})$, start reading $T_{L_{col}-2}$;
5. Iterations $2 \leq i \leq \min[I_{max}, L_{col}]$. The column numbers correspond to the contracted tensor network with $L_{col}^{(i)} = L_{col} - 2(i-1)$ columns
   a. vCP1—apply $\mathbb{C}(\mathcal{T}_1^{(i)}, T_2)$; start reading T_3;
   b. vCP2—apply $\mathbb{C}(\mathcal{T}_{L_{col}^{(i)}}^{(i)}, T_{L_{col}^{(i)}-1})$; start reading $T_{L_{col}^{(i)}-2}$;
6. If $L_{col} \leq I_{max}$, carry out vertical compression of the "big tensor" and finish;
   a. Apply $\mathbb{V}(T^{(L_{col})}$;
   b. Write result;
   c. Kill the instance;
7. Else, prepare the data for the Stage 2 algorithm;
   a. vCPU1—save $\mathcal{T}_i^{(i)}$;
   b. vCPU2—save $\mathcal{T}_{L_{col}-i}^{(i)}$;
   c. Kill the instance.

Stage 2 TNC algorithm. This stage starts with a tensor network $\mathcal{T}^{(I_{max})}$ with $2(L_{col} - I_{max})$ columns and L_{row} rows. Multiple partial contractions will be done for each column of $\mathcal{T}^{(I_{max})}$ during this stage.

The number of vCPUs for the instance used for successive iterations may increase. Results of a partial iteration have to be saved at the end of the partial iteration. The parameters for this phase are:

- $\mu_{pc}^{(i)}$—the space per tensor required for partial contraction at iteration i

$$\mu_{pc}^{(i)} = \mu + \mu^{(i-1)} + \mu^{(i)}; \tag{A.32}$$

  partial contraction increases the space required by each tensor;
- $\mathbb{C}_{pc}\left(\mathcal{T}_k^{(i)}, T_j, s\right)$—partial contraction operator applied to segment s of columns $\mathcal{T}_k^{(i)}$ and T_j in Stage 2;
- $n_r^{(i)}$—number of rows of a column segment for each partial contraction at iteration i given by

$$n_r^{(i)} = \left\lceil \frac{m}{\mu_{pc}^{(i)}} \right\rceil. \tag{A.33}$$

- $p^{(i)}$—number of partial contractions per column at iteration i;

$$p^{(i)} = \left\lceil \frac{L_{row}}{n_r^{(i)}} \right\rceil. \tag{A.34}$$

  The total number of partial contractions at iteration i is $2p^{(i)}$;
- The number of vCPUs for iteration i is

$$N^{(i)} = 2p^{(i)}. \tag{A.35}$$

- $L_{col}^{(i)}$—the number of columns at iteration i of Stage 2;
- $I_{Max} = L_{col} - I_{max}$—the number of iterations of Stage 2 assuming that Stage 3 is not necessary;
- $\mathbb{A}_{pc}\left(\mathcal{T}_{k,p^{(i)}}^{i}\right)$—assembly operator for the $p^{(i)}$ segments resulting from partial contraction of column k at iteration i.

Stage 2 TNC consists of the following steps:

1. For $i = 1, I_{Max}$
  a. Compute $\mu_{pc}, n_r^{(i)}, p^{(i)}, N^{(i)}$;
  b. If $N \leq 40$, start an instance with $N = N^{(i)}$; else, start multiple C4.10xlarge instances to run concurrently all partial contractions;
  c. For $j = 1, p^{(i)}$
    i. $vCPU_j$
      - Read $\mathcal{T}_{1,j}^{(i)}$ and $T_{2,j}$ and apply $\mathbb{C}_{pc}\left(\mathcal{T}_{1,j}^{(i)}, T_{2,j}\right)$,
      - Store the result $\mathcal{T}_{1,j}^{(i+1)}$;

    ii. $vCPU_{j+p^{(i)}}$
      - Read $\mathcal{T}_{L_{col}^{(i)},j}^{(i)}$ and $T_{L_{col}^{(i)}-1,j}$ and apply $\mathbb{C}_{pc}\left(\mathcal{T}_{L_{col}^{(i)},j}^{(i)}, T_{L_{col}^{(i)}-1,j}\right)$,
      - Store the result, $\mathcal{T}_{L_{col}^{(i)},j}^{(i+1)}$;

  d. Assemble partial contractions
    i. $vCPU_1$
      - Apply $\mathbb{A}_{pc}\left(\mathcal{T}_1^i, p^{(i)}\right)$,
      - Store $\mathcal{T}_1^{(i+1)}$;

    ii. $vCPU_2$
      - Apply $\mathbb{A}_{pc}\left(\mathcal{T}_{L_{col}^{(i)}}^{i}, p^{(i)}\right)$,
      - Store $\mathcal{T}_{L_{col}^{(i+1)}}^{(i+1)}$.

2. If $i < I_{Max}$, proceed to the next iteration, $i = i + 1$; else,
  a. Apply $\mathbb{V}\mathcal{T}^{(I_{Max})}$;
  b. Write TNC result;
  c. Kill the instance.

Stage 3 TNC algorithm. The algorithm is similar with the one for Stage 2, but now a single tensor is distributed to multiple vCPUs.

An analysis of the memory requirements for TNC. Let us assume that we have L tensors per column, and each tensor has dimension D. Consider the leftmost, or equivalently the rightmost column, and note that the number of bonds differs for different tensors; the top and the bottom tensors have two bonds,

and the other $L-1$ have $three$ bonds, so the total number of elements in this column is

$$\mathcal{N}_1^{(0)} = 2D^2 + (L-2)D^3. \tag{A.36}$$

The top and bottom tensors of the next column have three bonds, and the remaining $L-2$ have four bonds; thus, the total number of elements in the second column is

$$\mathcal{N}_2^{(0)} = 2D^3 + (L-2)D^4. \tag{A.37}$$

After contraction, the number of elements becomes

$$\mathcal{N}_1^{(1)} = 2D^3 + (L-2)D^5. \tag{A.38}$$

Each tensor element requires two double-precision floating-point numbers, thus the amount of memory needed for the first iteration is

$$\begin{aligned} \mathcal{M}^{(1)} &= 2 \times 8 \times [D^2 + (L-2)D^3 + 2D^3 + (L-2)D^4 + 2D^3 + (L-2)D^5] \\ &= 16 \times [2D^3 + (L-2)D^4 + 2D^2(1+D) + (L-2)D^3(1+D^2)] \end{aligned}. \tag{A.39}$$

The amount of memory needed for iterations 2 and 3 are

$$\begin{aligned} \mathcal{M}^{(2)} &= 16 \times [2D^3 + (L-2)D^5 + 2D^3 + (L-2)D^4 + 2D^4 + (L-2)D^7] \\ &= 16 \times [2D^3 + (L-2)D^4 + 2D^3(1+D) + (L-2)D^5(1+D^2)] \end{aligned} \tag{A.40}$$

and

$$\begin{aligned} \mathcal{M}^{(3)} &= 16 \times [2D^4 + (L-2)D^7 + 2D^3 + (L-2)D^4 + 2D^5 + (L-2)D^9] \\ &= 16 \times [2D^3 + (L-2)D^4] + 2D^4(1+D) + (L-2)D^7(1+D^2)) \end{aligned}. \tag{A.41}$$

It follows that the amount of memory for iteration i is

$$\mathcal{M}^{(i)} = 16 \times [2D^3 + (L-2)D^4 + 2D^{i+1}(1+D) + (L-2)D^{2i+1}(1+D^2]. \tag{A.42}$$

When $D = 20$ and $L = 100$, the amount of memory for the first iteration is

$$16 \times [2 \times 20^3 + 98 \times 20^4 + 2 \times 20^2 \times (1+20) + 98 \times (20^3 + 20^5)] = 5\,281\,548\,800\ bytes. \tag{A.43}$$

This example shows why only the most powerful systems with ample resources can be used for TNC. It also shows that an application has to adapt, the best it can, to the packages of resources provided by the CSP, while in a better world, an application-centric view should prevail, and the system should assemble and offer precisely the resources needed by an application, neither more nor less.

A.7 A simulation study of machine-learning scalability

Training Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs) is computationally intensive. In Section 13.2, we discussed results reported by Google researchers showing that

FIGURE A.16

StarCraft is a real-time strategy game played against one opponent. Each player has hundreds of game pieces.

training a CNN with 89 layers, 100×10^6 weights, and 1 750 operations/weight required 17.5×10^9 operations. The training set size for activity recognition for an eight-layer DNN required 10^6 videos, and the training time was reported to take some 30 days.

We posed to students in a graduate cloud computing class the question if it is feasible to use machine learning for predicting the computational effort required to train a DNN. A CNN used to predict the computational effort for DNN training should have as input data describing the DNN and data describing the computational effort for training a range of DNNs. CNN output should be the training time, and the number of arithmetic operations required for DNN training.

DNN data should include: hidden layer count; number of neurons in each hidden layer; number of neurons in input and output layers; size of the training set; learning rate; number of epochs; and the loss function determining how far a given output is from correct value. System monitoring data includes training time; number of different arithmetic operations; CPU utilization; number of messages exchanged; number of I/O operations; etc.

A preliminary study conducted in Spring 2020 by Thomas C. investigated the scalability of ML training function of DNN characteristics. In each experiment only the computation time was recorded. The project expanded on work done for a machine learning class to control game pieces in a *StarCraft* game; see Fig. A.16. The ML project had three phases: (i) build a dataset by running a specific game scenario 20 000 times; (ii) train the model using to predict a "best" action; and (iii) rerun game scenarios using the new predicted *best* action.

The new project was to simulate scaling up the workload to 10, 100, or 1 000 machines. The project used several AWS services including EC2, S3, SimpleDB, Elastic Beanstalk, SNS, Lambda, and API gateway to create a system of multiple AWS instances for running ML workloads and then to test scaling using a combination of real and simulated workloads. Each training dataset was created by running *StarCraft* with game pieces in a certain positions and included the input location of game pieces for the player and the opponent and an integer reflecting the battle score—a high battle score

FIGURE A.17

Each record included: (i) player units; (ii) opponent units; and (iii) battle score.

meaning that the battle went well for the player and a low or negative score if the battle did not go well for the player. The data set size reflected the number of records in the training data set; see Fig. A.17; the project used datasets with 4k, 6k, 8k, and 10k records.

Other training parameters are: (i) batch size, i.e., how many records to train on before updating the weights (64, 128, 256); (ii) the number of epochs, i.e., how many times the dataset is used to train the neural network (125, 250, 500, 1 000); and (iii) the loss function.

System organization is shown in Fig. A.18. *Model workload clients* perform several functions: (i) train models using seven training parameters: data set size, learning rate, hidden layer count, hidden layer size, batch size, number of epochs, and loss function; (ii) get training parameters using AWS HTTP Endpoint; and (iii) upload results including training time, model loss, and model accuracy. *Data set workload clients* generate and upload datasets to S3 buckets and upload the results.

The *workload coordinator* uses the AWS Lambda service to get training parameters and data set metadata and then upload the model and the datasets. It also stores three DynamoDB tables, the coordinator state, models, and datasets.

Simulation experiments created four datasets with a total of 40 000 records and trained 5 000 models. The average training time was 143.7 s with a standard deviation of 250.2 s. Fig. A.19 shows a histogram of learning models function of training time and learning rate, respectively. As expected, the training time increases nearly linearly with the number of hidden layers and with the size of hidden layers; see Fig. A.20.

Training time decreases as the batch size increases; see Fig. A.21 (left); accuracy increases as the number of epochs increase; see Fig. A.21 (right). The results of a scaling experiment using a mock workload are summarized in Fig. A.22. The number of agents, i.e., servers involved increases linearly, and the number of requests increases faster than linearly as the time progresses until the experiment was stopped when reaching the Lambda free-tier limit.

A.8 Cloud-based task alert application

TaskAlert application developed by Brandon G. is designed to manage a variety of user tasks, some involving computer-based activities and tasks related to any other types of professional or personal activities. The application allows users on client systems access to a set of microservices performing

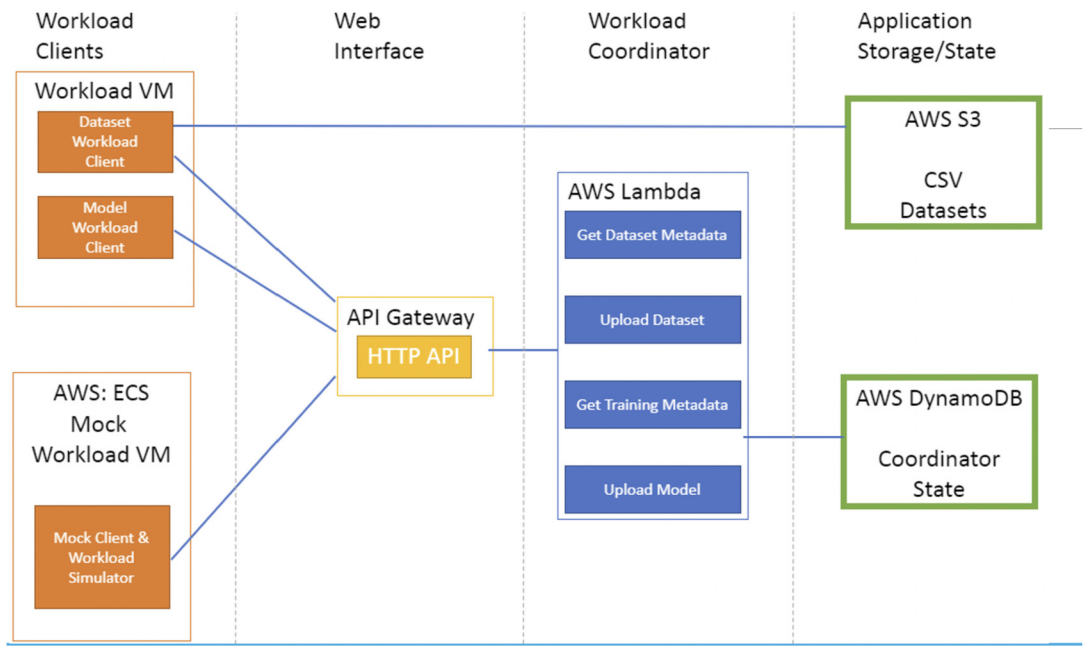


FIGURE A.18

System organization. There are two types of clients, *data set workload* to create datasets and *model workload* to train models. A *workload coordinator* assigns workload metadata and stores results. The *mock workload client* simulates workload clients.

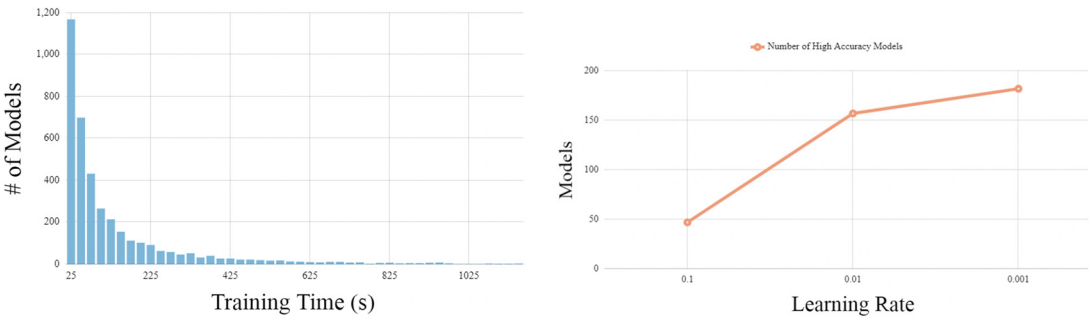


FIGURE A.19

Number of models function of training time and learning rate.

functions such as: (i) access a list of tasks to be completed; (ii) manage account information and register for the service; (iii) supply task information, to automatically create a task in response to an email sent to the application.

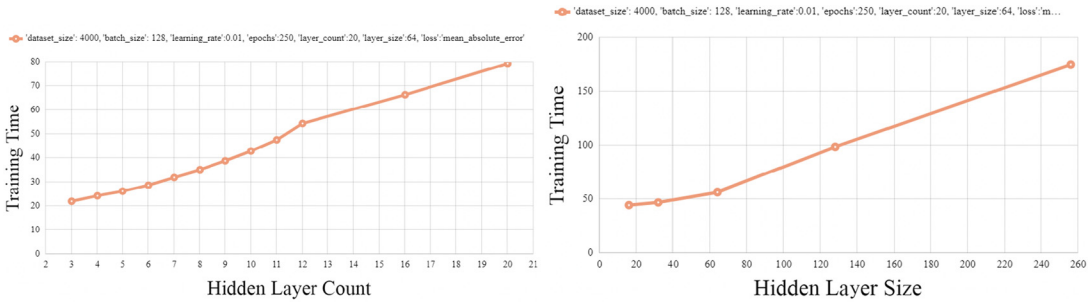


FIGURE A.20

Training time function of hidden layer (a) count; (b) size.

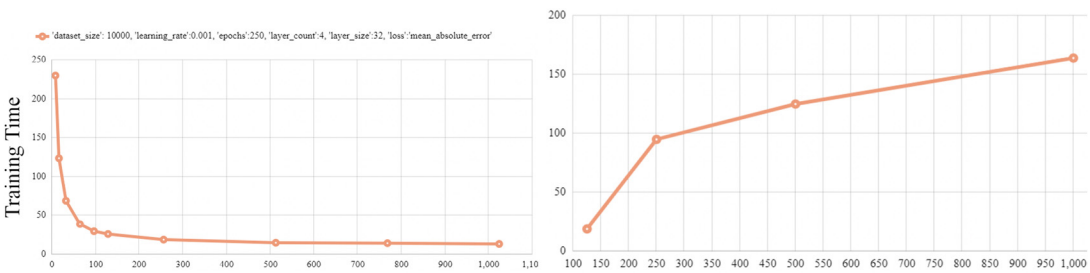


FIGURE A.21

Training time function of batch size and accuracy function of number of epochs.

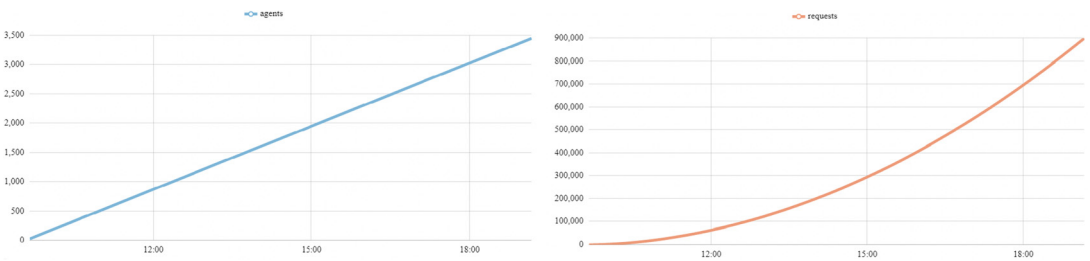


FIGURE A.22

Scaling experiment using a mock workload. The experiment started at 9 am and was stopped after some 10 h after reaching AWS Lambda free-tier limit when 3 500 agents and 900 000 requests had been processed without any errors.

Task description identifies the task type, resources needed for task completion, deadlines, milestones, task priority, task dependencies, events and/or conditions that could affect task completions, interactions with other individuals required for task completion, and possible conflicts with other tasks.

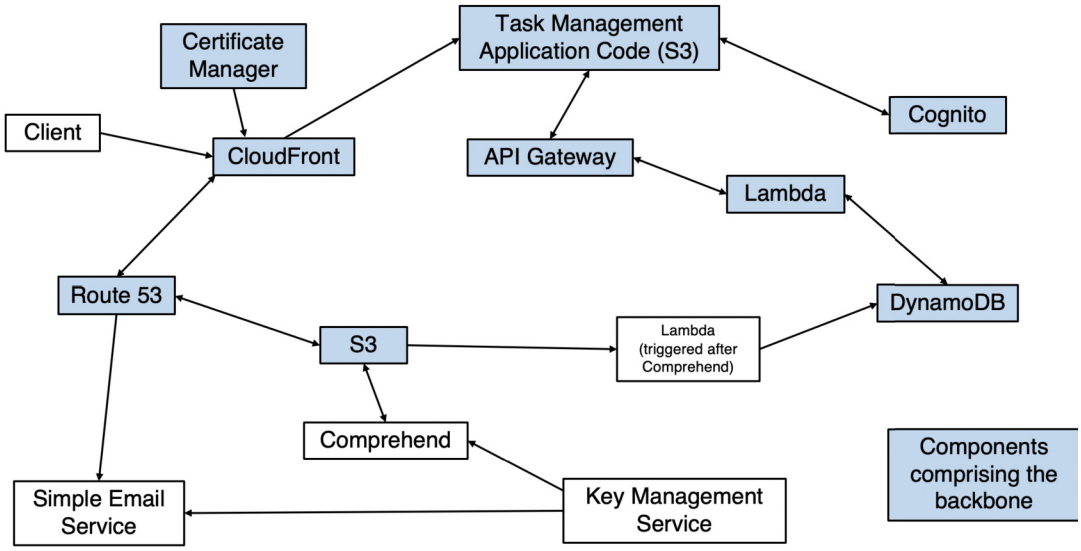


FIGURE A.23

TaskAlert organization and AWS services used by the application.

Devices used to communicate with *TaskAlert* and the communication means; email, messaging, and phone alerts are also part of the task description.

Application front end hosts a user interface (UI) allowing authorized and authenticated users to interact with the back end; see Fig. A.23. The front end is implemented in Javascript with Node.js and React, a JavaScript library used to create interactive UIs. Several other libraries are used by the application front end: (i) *React-router-dom* library used to route users to the proper pages; and (ii) *Axios* library used to access custom REST API to interact with other AWS services. The *AWS Amplify* framework is used to access *Cognito* API to support user accounts. The application uses extensively the Lambda service, which runs user code without the need to provision or manage servers. Moreover, the user pays only for the compute time consumed by the application; it charges for every 100 ms when the code is triggered.

TaskAlert's back end supports task creation, deletion, modification, task retrieval, and maintaining user accounts using *Cognito.* Other AWS services used by *TaskAlert* are:

1. *Route 53*—allows users to access the service through *mytaskalert.com* domain.
2. *CloudFront*—converts user HTTP requests to HTTPS requests to *mytaskalert.com.*
3. *Certificate Manager*—issues a TLS/SSL certificate for *mytaskalert.com.*
4. *S3*—hosts the application code and dependences for *mytaskalert.com*
5. *API Gateway*—used by the application code to trigger the appropriate *Lambda* functions.
6. *Lambda*—performs functions specified by API Gateway to create, modify, delete, and retrieve user tasks.
7. *DynamoDB*—stores user tasks and allows task modification, deletion, and retrieval.

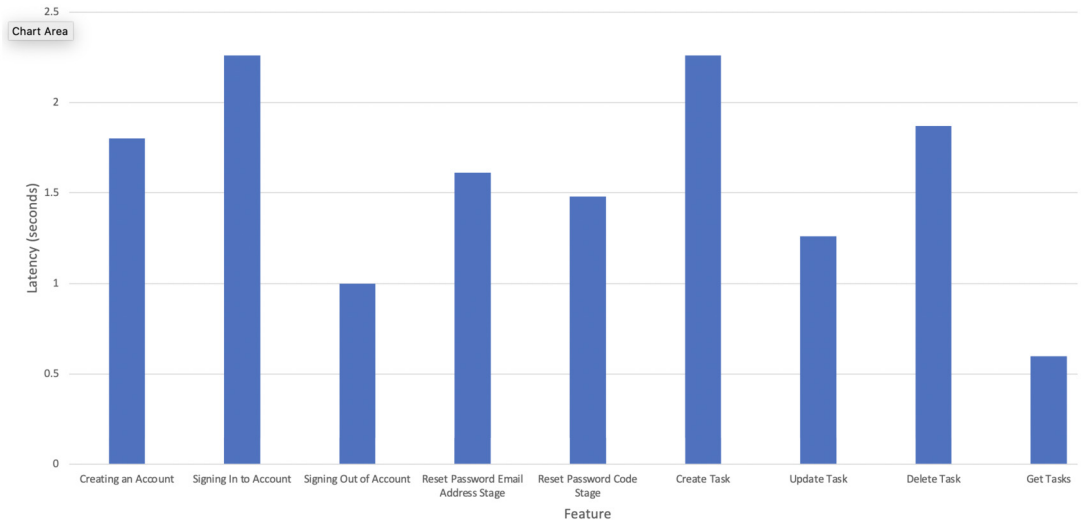


FIGURE A.24

TaskAlert execution profile. The time in seconds for various task management activities.

8. *Cognito*—allows users to sign in through social identity providers.

Fig. A.24 shows the time for various task management activities. The response time for task creation and signing into an existing account is about two s, while the response time for other task management activities is around one second.

A.9 Cloud-based health-monitoring application

The cloud-based health-monitoring application, developed by Mojtaba T., is designed to monitor outpatients who need constant supervision, individuals with chronic health problems, high-risk individuals, and frail elderly people. The system uses smartphones and wearable sensors to collect monitoring data from patients, as shown in Fig. A.25.

Existing health-monitoring systems seldom provide extended physiological data and a fast response time in case of emergencies. The cloud-based system takes advantage of a wide range of sensors and high-bandwidth wireless technology to connect patients with healthcare providers, emergency services, pharmacies, rehabilitation centers, and other relevant facilities. The system enables healthcare providers to exchange information, medical records and test results and to consult with each other for the benefit of a patient; it also provides access to healthcare databases.

Physiological and behavioral data is collected from a range of sources including: personal electrocardiogram (ECG) devices, electroencephalograph (EEG) headsets, photoplethysmographs (PPG), accelerometers, gyroscopes, and gait-cycle monitors. Photoplethysmography (PPG) is a optical technique used to detect volumetric changes in blood in peripheral circulation. A gait cycle is the time

FIGURE A.25

Physiological and behavioral data is collected from a range of sources.

period or sequence of events or movements from the instant one foot contacts the ground until the foot contacts the ground again and involves propulsion of the center of gravity in the direction of motion.

Some data is collected periodically, e.g., on hourly or daily bases, and others only when sensors signal out-of-range measurements. The vital signs of outpatients and other data such as gait cycles are streamed continually. It follows that the system must be nimble and in some cases provide real-time response.

The main functions of the system are: (i) user authentication; (ii) access control; (iii) data synchronization across multiple devices; (iv) patient behavior analysis; (v) data storage management; (vi) data collection from various sources; (vii) real-time data collection and response.

The application needs a more agile and flexible storage and uses AWS Lake Formation to set up a secure data lake. A data lake is a centralized, curated, and secured repository storing structured and unstructured data at any scale; see Fig. A.26(a). Data can be stored as is, without having first to be structured, and can be used for multiple types of analytics from dashboards and visualizations to big data processing, real-time analytics, and machine learning; see Fig. A.26(b).

The applications uses AWS Kinesis system for data streaming from multiple sources to multiple EC2 instances and then stores the results of processing kinesis streams to data lakes, as shown in Fig. A.27. Amazon Kinesis service supports process streaming data at any scale, along with the flexibility to choose the optimal processing for every data stream, including real-time data provided by various health-monitoring sensors.

The application is designed to collect, preprocess, analyze, and store Big Data produced and consumed by a large user population. To accomplish its mission, the application must rely on a powerful framework, and it was decided to use Spark. Recall from Section 4.12 that Apache Spark is an open-source Big Data processing framework used to manage Big Data processing for various data types.

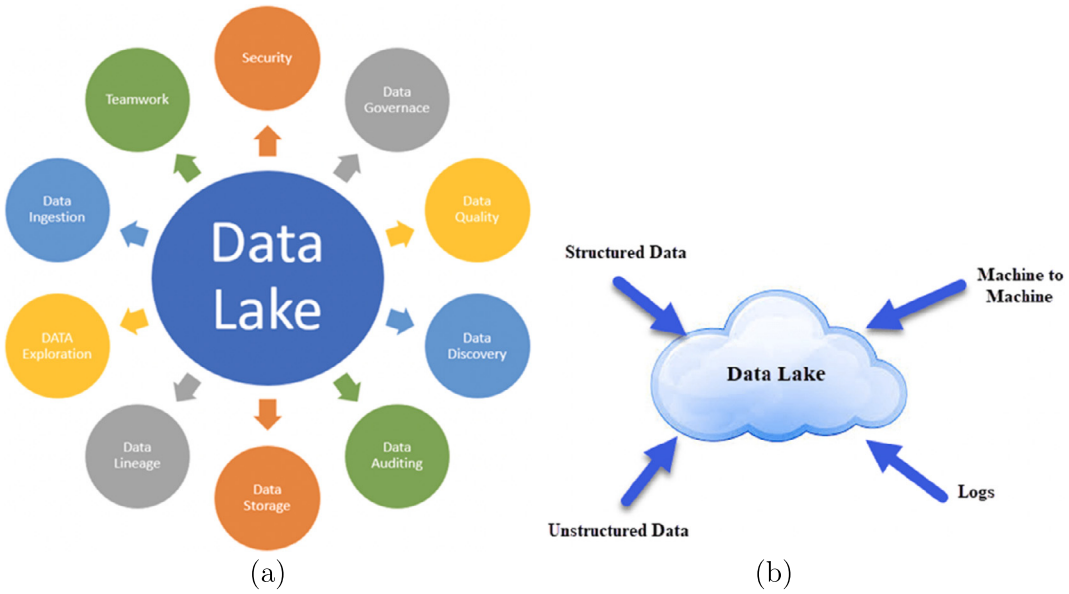


FIGURE A.26

(a) Multiple functionality supported by data lakes. (b) Various types of data can be stored in data lakes.

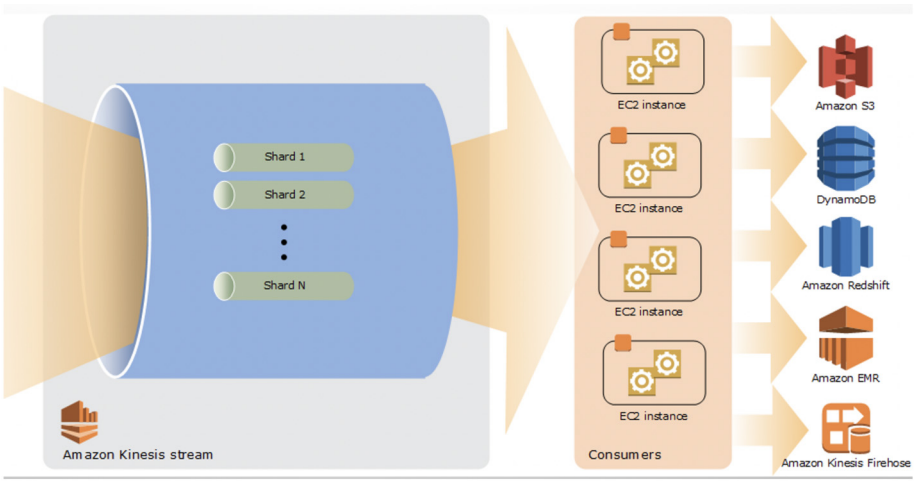


FIGURE A.27

The flow of data from multiple sources through the AWS Kinesis to EC2 instances for processing and to data lakes for storing measurement results and raw data.

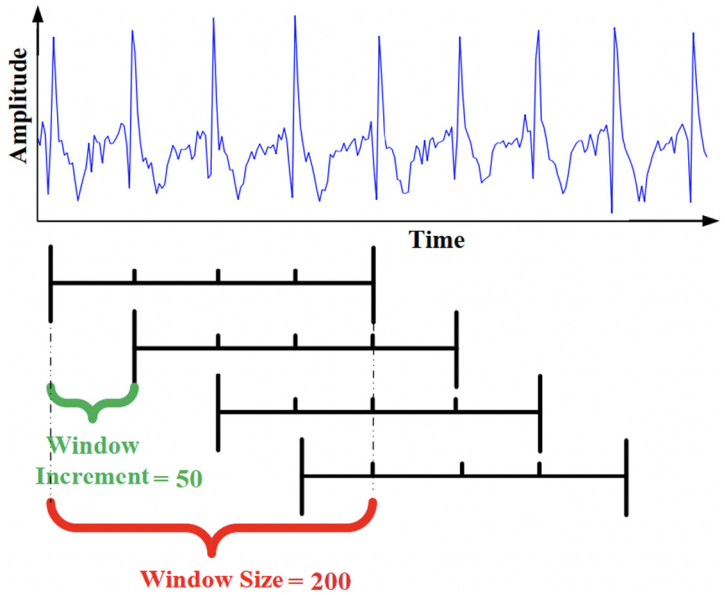
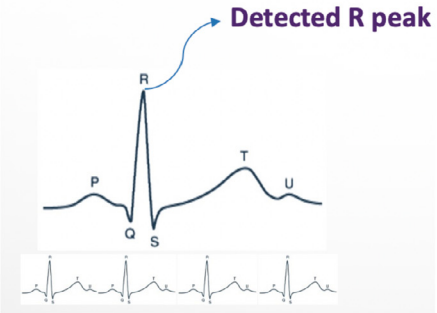


FIGURE A.28

EEG preprocessing. A band-pass filter for 0.5–4 Hz removes low-frequency power-line noise and high-frequency noise and segments the EEG signal to create 200 data-point windows overlapping each other on 50 data-point window.

	Feature	Unit	Description
Time – Domain	HRV		Heart Rate Variability
	AvgHR	bpm	Average Heart Rate
	MeanRR	ms	Mean of selected R-R series
	NN50	count	Number of consecutive R-R intervals that differs more than 50 millisecond
	SD_HR	1/min	Standard Deviation of Heart Rate
	SD_RR	1/min	Standard Deviation of R-R interval
	RMSSD	ms	Root Mean Square of the differences of selected R-R interval series
	SE	-	Sample Entropy
Frequency-Domain	PSE	-	Power Spectral Entropy

(a)



b)

FIGURE A.29

(a) EEG features in time-domain and in frequency-domain. (b) R-peak detection.

Along with MapReduce, it also supports SQL queries, streaming data, graph data processing, and machine learning techniques.

To illustrate the complexity of data analysis, we show the preprocessing and feature extraction for EEG in Figs. A.28 and A.29. ECG peak detection is used to detect the combination of Q, R, and S waves or the so-called QRS complex. The QRS complex is a peak model for ECG signal including Q-valley point, R-peak point, and S-valley point. Other important peak points in the ECG signal are the P-peak point and the T-peak point. The detection of the QRS complex is critical for ECG signal processing.

A band-pass filter for 0.5–4 Hz removes low-frequency powerline noise and high-frequency noise and segments the EEG signal to create 200 data-point windows overlapping each other on 50 data-point window. Fig. A.29(a) shows the EEG features. For each segment of the data, various features are extracted and results are saved in a file. Fig. A.29(b) illustrates the R-peak detection.