Design and Implementation of Utility-Based Radio Resource Optimization Using CAPRI

Krisakorn Rerkrai⋆, Jad Nasreddine⋆, Zhou Wang†, Janne Riihijärvi⋆, and Petri Mähönen⋆
⋆ Institute for Networked Systems, RWTH Aachen University, Germany
† European Microsoft Innovation Centre, Germany
Email:{kre}@inets.rwth-aachen.de

***Abstract*—In this paper we introduce a utility-based optimization approach for wireless communication based on developed Common Application Requirement Interface (CAPRI). The proposed system allows applications to register their utility functions to a radio resource controller at run-time and in automatic way. This enables the implementation of utility-based optimization and run-time reconfiguration of applications. Specifically, in contrast to traditional utility-based optimization approaches that usually aim at optimizing a single application deployed for the entire system, our approach aims to generalize and express the requirements of multiple applications. To validate our proposed system, we have implemented a prototype of CAPRI using Windows platform. The implementation has been made publicly available under an open source license. We show that the overhead induced by the adoption of the CAPRI framework in terms of memory and processing requirements is small, and additional latencies induced by the framework are negligible. Further, a practical case study of multi-application utility-based optimization has been addressed and the results have been analyzed.**

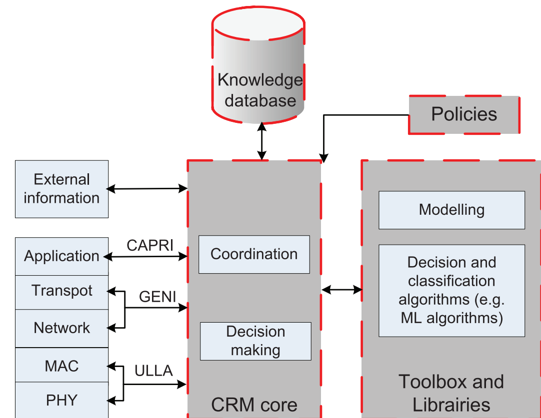

Fig. 1. CRM architecture

I. Introduction

The resource management in wireless communications is a crucial task due to the fact that the radio environment is very hostile in terms of fast and unpredictable changes, uncontrollable interference and scarcity of resources. The radio resource management has become more and more sophisticated in wireless systems. During the last decade research community has started to consider also increasingly, and complex cross-layer optimization methods as a part of resource management.

We have earlier proposed in the context of cognitive radios a new architecture for self-optimizing wireless systems. The key aspect in this architecture is Cognitive Resource Manager (CRM) framework [1], [2] that is a cognitive radio extension of traditional Radio Resource Managers (RRM). Through the principles of modularity, run-time reconfigurability, open interfaces and open policy language, the CRM framework enables an easy implementation of cross-layer optimization, complex control and learning mechanisms.

A high level architecture of the CRM framework [3] is depicted in Fig. 1. In this framework the toolboxes and libraries provide a collection of algorithms, components and tools that can be used to optimize the resources while satisfying application requirements. In order to enhance system performance, the toolboxes and libraries use the knowledge database that stores the acquired long-term and short-term information from the environment and the network. In addition, a policy layer is used to manage the rules and policies set by different stakeholders and resolve any conflict between them. The managed policies are not only regulatory policies (e.g. spectrum use rules) but can also be user preferences, device manufacturer configurations, and operator specific policies.

The CRM core acts as the kernel component of the system that facilitates the construction and the management of all the above mentioned components and their interactions with the environment and the protocol stack. The interaction of the CRM and its components with the protocol stack is enabled by the integration of open interfaces that facilitate also the implementation of cross-layer and utility-based optimization techniques. These interfaces are ULLA (Universal Link Layer APIs) [4], GENI (Generic Network Interface) and CAPRI (Common Application Requirement Interface) [5], [6]. ULLA is used for interaction with the link/physical layer, GENI for interaction with the transport/network layer and CAPRI for interaction with the application layer. In this paper we focus on explaining the use of CAPRI to enable utility based optimization. We report specifically on the CAPRI reference implementation in Windows OS and analyze its performance.

The remainder of this paper is organized as follows. We discuss the motivation of our work in Section II. In Section III we describe the CAPRI framework architecture, implementation and performance evaluation. Later we discuss how our

978-1-4244-9538-2/11/$26.00 ©2011 IEEE

optimization algorithm works in Section IV. We then provide a practical example in Section V. We finally conclude the paper in Section VI.

II. Enabling multi-application utility-based optimization

The success of wireless communication in attracting a wide range of users has motivated the companies to develop a plethora of applications that have different characteristics. Therefore, these applications have different requirements on low-level parameters of the protocol stack such as delay, throughput, and jitter. For instance, VoIP applications have tight requirements on delay and packet error rate whereas streaming applications are more sensitive to jitter. However, the actual implementations of RRM do not take into account this diversity directly or provide a set of predefined requirements (e.g. UMTS QoS classes) since there is no interface that allows the applications and individual flows to express dynamically and more precisely their requirements and preferences in terms of network constraints and performance. A solution for this problem is to define an API that registers the different applications together with their utility functions in the entity responsible of performing the resource management. In this context, CAPRI is proposed as an interface between the CRM core and the application layer to allow applications to register and update their requirements and preferences in terms of network constraints and performance. This way, we are not bounded by any specific preferences, as we have functional descriptions within the architecture. In addition, when CAPRI is enabled, it allows also the resource manager to control some tunable parameters of the applications. The integration of this interface in the CRM framework will allow the implementation of utility- and policy-based optimization techniques. It should be noted that CAPRI is not dependent on the presence of CRM. It can be also used with any other resource manager if the data format is respected.

To the best of our knowledge, most of the practical works on utility-based optimization have focused on solving the optimization problem for a single application or considered a single requirement for the whole system [7], [8]. In contrast, our approach based on the proposed CAPRI framework, tries to generalize and express the application requirements to the network. The proposed approach can be used to solve a multi-objective optimization problem. However, it is not trivial to combine these requirements into a single expression which can represent the total utility in order to perform the optimization process. In particular, we take into account that several applications can run on the same link and share resources, while each application can generate several data flows to/from various destinations/sources. Moreover, we consider that data flows have different levels of priority that can be set by the administrator of the network or the user. In order to formulate this relation, we use a simple weighted sum approach based on objective weighting presented in [9]. The weighted sum approach attempts to maximize the sum of the positively normalized, weighted, single objective scores of the parameter set solution. Hence, we define a node utility U_{node} by prioritizing the data flows:

$$U_{node} = \sum_{i=1}^{N} p_i \cdot U_{flow,i}(\mathbf{a}_i), \quad (1)$$

where $U_{flow,i}$ is the utility function of data flow i having a nonnegative priority p_i (high p_i for high priority), $\mathbf{a}_i = (a_{i,1}, ..., a_{i,K})$ is a K-dimensional vector of network attributes characterizing a connection used by flow i and N is the number of data flows running on the node. It should be noted that the summation of priorities is one.

Similar to U_{node} and as an illustrative example, we use in our implementation a particular separable utility function defined by

$$U_{flow,i} = \sum_{j=1}^{K} w_{i,j} \cdot U_{i,j}(\mathbf{a}_{i,j}), \quad (2)$$

where $U_{i,j}(\mathbf{a}_{i,j})$ is a utility function related to a set of network attributes $\mathbf{a}_{i,j}$, $w_{i,j}$ is a weighting coefficient of $U_{i,j}$ and $\sum_{j=1}^{K} w_{i,j} = 1$. The definition of $U_{i,j}$ and the value of $w_{i,j}$ are application-dependent and can be provided by the application developer or a third party. These utility functions are normalized to have values in the range [0,100]. As a result, the flow utility value is in the same range of [0,100] so that each flow has the same range making any comparison between flow utilities possible.

In typical optimization problems, the resource manager attempts to perform optimizations at physical, MAC, or transport layer, or to perform optimizing across multiple layers. Nevertheless, in some circumstances the optimizers are bounded with physical conditions and cannot perform optimizations in those layers. For example, when a wireless device is surrounded with many other wireless devices, it is impossible or nearly impossible to switch to another channel that is occupied by other wireless devices due to high interference. The other alternatives like increasing the channel width are also not appropriate in such a situation due to the interference that can be generated in this case. Alternatively, the optimization can be done at the application level by tuning application parameters during run-time.

There are several methods to change the configurations of the applications. For instance, adaptive video streaming applications like dynamic streaming in Adobe Flash Player [10] and IIS Smooth Streaming for Microsoft Silverlight [11] vary video quality and packet sizes during playback so that the available bandwidth and local CPU processing power can support. However, all the existing methods are implemented independently in the application itself and do not have full knowledge on network and radio resource utilization and tradeoffs. This might lead to suboptimal solution since the two optimizers work independently. Therefore, we argue that the use of CAPRI will enhance the performance of the system since it allows information exchange between the applications and the resource manager.

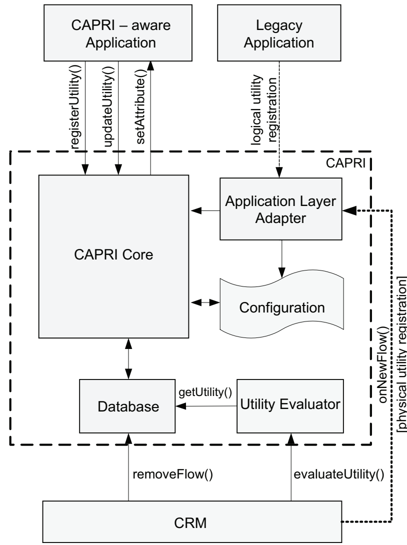

Fig. 2. Implementation architecture of the CAPRI framework.

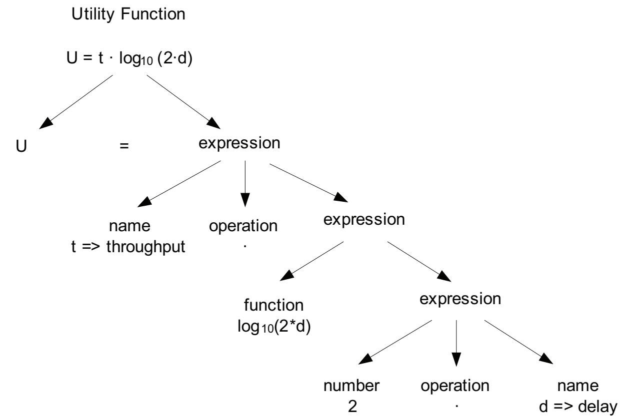

Fig. 3. Evaluation tree construction.

III. System architecture and implementation

A. CAPRI framework: design and implementation

The Common Application Requirement Interface (CAPRI) enables applications to express their requirements and preferences in terms of network constraints and performance, which can be subsequently used by the resource manager in their utility-based optimization process. In addition, CAPRI allows applications to update their requirements during runtime. By using CAPRI, the application performance can be assessed through the computation of utility functions. Each application is associated with a utility function that is communicated to the CRM through CAPRI and that specifies how its performance should be evaluated.

The architecture of CAPRI framework is depicted in Fig. 2. The framework includes CAPRI Core, an Application Layer Adapter (ALA), a database, and a utility evaluator. The main component is CAPRI Core. Its functionality includes parsing utility functions, processing and sending commands from/to applications and CRM, registering and deregistering applications, and parsing the configuration. The role of ALA is to inspect all the new data flows detected by CRM in order to seamlessly adapt legacy applications to fit within the CRM framework by abstracting away any application specific details. The utility evaluator is the component responsible of computing the utility values of the different applications based on the instantaneous attributes provided by the CRM and the registered utility functions through CAPRI Core.

The CAPRI Core provides two ways for applications to register and update utility functions.

- CAPRI-aware applications can register themselves through the CAPRI Core. However, the registration will be completed once the actual data flow is detected by CRM and then ALA indicates the flow information to the CAPRI Core. Moreover, the CAPRI Core also provides to these applications commands that are used to update utility functions and to configure some application attributes such as maximum allowable bandwidth and application level options.
- Legacy applications that are not developed with a CAPRI interface and library, can be automatically registered by CRM by loading predefined utility functions from a configuration file according to an application type. Currently, the application type is determined by network ports an application is using. This operation is performed within ALA. The utility functions of legacy applications can be updated by overwriting the utility functions in the configuration file. The CAPRI Core will be notified if the configuration file has been modified.

B. Utility evaluation

In CAPRI, the utility function is parsed by using lex/yacc [12] to build a utility evaluation tree. The parser applies the scalar expression rules defined in the language repeatedly. This way, yacc can parse recursive rules very efficiently. For example, Fig. 3 shows how to parse a specific utility function defined by $U = t \cdot \log(2 \cdot d)$. Naturally, a yacc parser does not construct this tree as a data structure. Instead, we have to dynamically allocate memory for each constructed vertex in the tree. The tree is constructed bottom-up. There are three possible vertex patterns, which include a value (number), an attribute (name) and an operator. Functions, such as log() or exp(), are processed as an operator with one input parameter. After the tree is built, it is stored in the database together with a corresponding flow identifier (FID) consisting of five attributes: protocol, IP address, source address, destination address, transport source port and transport destination port. C++ STL map container is used to map a given FID to a parsed utility tree.

The utility evaluator evaluates utility functions based on the information collected by the CRM. When *evaluateUtility()* is called by CRM, the utility evaluator uses the FID as a key to find the associated evaluation tree. Utility evaluation is fundamentally a reverse ordering of a constructed tree. The

originally allocated vertices are visited first. This results in operations being applied in the order that they were encountered during parsing. We also apply recursive rules with the tree reconstruction. If a vertex type is TYPE_VALUE, then it returns a value. If a vertex type is TYPE_ATTRIBUTE, then it returns a mapped value of a given attribute. If a vertex type is TYPE_OPERATION, it will recursively call a function *evaluateUtility()* in order to retrieve values of the operands.

To evaluate application performance, CRM calls *evaluateUtility()* by providing FID together with attribute values. The utility evaluator in the CAPRI framework finds the utility tree from utility table using the FID. Then it computes U_{flow} by considering different network performance indicators like delay (d), throughout (t), packet loss (l) and jitter (j), and their weights that are provided by the application. Using (2) this flow utility can be written as

$$U_{flow} = w_t \cdot U_t(t) + w_l \cdot U_l(l) + w_d \cdot U_d(d) + w_j \cdot U_j(j), \quad (3)$$

where $U_t(t)$, $U_l(l)$, $U_d(d)$, and $U_j(j)$ are utility functions of throughput, packet loss, delay, and jitter, respectively whereas w_t, w_l, w_d, and w_j are the weighting coefficients of throughput, packet loss, delay and jitter, respectively. These coefficients satisfy the relation $w_t + w_l + w_d + w_j = 1$.

C. Interaction with CRM

This subsection explains how CRM interacts with the components in the CAPRI framework. For legacy applications, once CRM gets a notification from the protocol stack about a new flow created by an application, it then notifies the ALA together with the FID. The ALA loads a utility function from the configuration file based on the application type. Then it registers the utility function to the CAPRI Core. The ALA uses the application type to assign the corresponding utility function to the new flow. The default utility functions of legacy applications are defined in a configuration file, as the application is not aware of CAPRI. The utility function can be updated and automatically reloaded when the configuration file is modified. The CAPRI Core parses the utility function and constructs a utility tree, which is stored in the CAPRI database together with the identifier of the flow.

For CAPRI-aware applications, the application first registers its utility function. At the beginning, an FID of the newly registered application is not yet complete. It has only destination port, destination address, and protocol type. It does not have the full information until the actual data flow is initiated. Upon first data packet detection, the CRM notifies the ALA that a new flow is detected. The ALA checks if the destination port, destination address, and protocol type of the newly detected flow match with the recently registered incomplete FID in the database. If these parameters match with an existing entry in the database, the CAPRI Core will update that entry with an FID detected and notified from the CRM.

D. Performance evaluation

This subsection presents the evaluation of CAPRI framework implementation in terms of memory footprint and utility

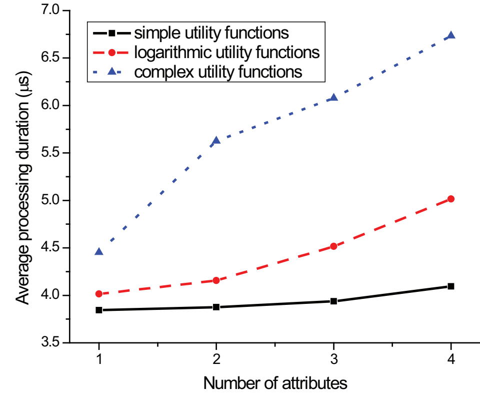

Fig. 4. Average processing duration of a single utility function.

evaluation duration in order to explore the tradeoff between flexibility and cost of implementation in terms of complexity and performance. The prototyping platform is based on two standard notebooks using an Intel Core2Duo clocked at 2.20 GHz and 2.0 GB of RAM. Altogether, the CAPRI Core, ALA, Database and CAPRI parser represent a static memory occupation of less than 3.5 MB. It should be noted that more than 60% of the memory is occupied by the use of C++ STL map container. Alternative techniques can be used to reduce the memory consumption.

We measured the utility evaluation duration for a single standard *evaluateUtility()* call. Several degrees of complexity of utility functions were also experimented. The results have shown that the average utility evaluation duration is around 65 μs including the time used to retrieve data from the driver. However, the latency introduced by parsing the utility functions is very low as shown in Fig. 4 where we measured the utility processing duration for a single standard *evaluateUtility()* call calculating a utility value with different number of attributes and different levels of complexity. We repeated the function call 1000 times for each trial. For example, when evaluating only one attribute the average evaluation duration is around 3.8 μs for a simple utility function, 4.0 μs for a utility function with a logarithmic function, and 4.5 μs for a complex utility function (with conditions and adjustable parameters). Therefore, the latency introduced by the CAPRI framework is acceptable for most of RRM technologies that do not require very fast decision making. As we can see this processing time is negligible compared to the total utility evaluation utility or the time needed to retrieve data. The latter, which is the order of tens of microseconds, is independent of the number of requested attributes.

IV. Optimization algorithm

Our resource optimizer determines the configuration actions based on utility values obtained from the utility evaluator. In this type of optimizers, the decision is usually not made immediately after the utility value drops. Instead, it is made when several consecutive conditions are met. This is so called *late decision making*. In principle, this is used to avoid transient

events that may occur and degrade U_{node} for a short period of time. For example, a short burst of high CPU processing load may affect data transmission from the video streaming server.

The proposed optimization process at the application level is triggered if one of the following conditions is fulfilled.

- U_{node} is lower than a utility threshold ($U_{node} < T_{Utility}$), where $T_{Utility}$ is an adaptive threshold over time, i.e. it changes a new value, which may be higher or lower, depending on changes of environmental conditions. In other words, this is a relative threshold at a particular time that reflects the actual environmental conditions. If the environmental conditions become worse, then $T_{Utility}$ gets lower, and vice versa.
- The difference between U_{node} and moving average utility value is greater than a given utility difference threshold ($|U_{node} - U_{avg}| > T_{UtilityDiff}$). If the difference is positive, then the optimizer should check if the resource can be optimized while keeping the current utility high. If the difference is negative, then the optimizer should find a solution to tune link parameters or application parameters. As a result, $T_{Utility}$ will be updated with a new value.
- The periodic timer times out. We use a watchdog mechanism to periodically check if the environmental conditions change over time. For instance, when we start an application, the optimizer tries to optimize the system with the best configuration to maximize U_{node}. After a certain time, U_{node} is still constant even if the environmental conditions improve. For instance, after a neighbor turns off its wireless devices, the adjacent channels are free to use. Without the watchdog, the optimizer is never triggered by the previous two conditions alone.

$T_{Utility}$ is updated to prevent frequent and unnecessary activation of the CRM decision mechanisms. For instance, if the environment conditions are bad for a long time but there is no better configuration than the actual one that leads to a utility lower than the threshold, the decision mechanisms will be activated very frequently and will take the same decision all the time. Therefore, by using the dynamic threshold, the CRM monitors the utility values and actions that have been taken, and sees if the utility is lower than the threshold while the CRM does not find a better solution. In this case the threshold will be lowered to prevent the frequent and unnecessary activation of the CRM. When the utility value starts to increase due to changes in the channel conditions or after the CRM finds new solutions, the threshold will be increased again to guarantee the best functionality of the system.

V. A PRACTICAL CASE STUDY

We shall give a simple example to show how the proposed algorithm works in practice. We have two applications in this example scenario: UDP video streaming (VLC) [13] as a legacy application and UDP download application (DL) as a CAPRI-aware application. The utility functions used in this example have been modified from [14] in order that the weighting coefficients and thresholds are suitable for our prototype. In our example we assume that VLC has higher priority than DL. This can be for instance due to the fact that the user prefers to give higher priorities to video streaming and therefore set the priorities accordingly. The following utility function is a utility function used for the DL application.

$$U_{DL} = 100 \cdot \frac{\log_{10}(min(t, BW_{max}) + 1000000) - 6}{\log_{10}(BW_{max} + 1000000) - 6}, \quad (4)$$

where t is a throughput measured in bit/s and BW_{max} can be defined by the application when registering to the CAPRI Core. Alternatively, BW_{max} can be adjusted during run-time by the CRM through the *setAttribute()* function.

The following utility functions is a utility function used for the VLC application.

$$\begin{aligned} U_{VLC} =& 0.2 * \left\{ 100 \cdot \frac{\log_{10}(t + 1000000) - 6}{1.491362} \right\} \\ &+ 0.8 * \{(100 - 0.4 \cdot d) \cdot [d < 5] \\ &+ 75 - 25 \cdot tanh(\frac{d - 12.95}{5}) \cdot [5 \leq d < 25] \\ &+(57 - (d \cdot 0.264)) \cdot [25 \leq d < 215]\}, \end{aligned} \quad (5)$$

where d is a delay measured in milliseconds.

To run the experiment, we perform the following steps:

1) Start VLC streaming at around 6 Mbps.
2) Start DL with BW_{max} of 9 Mpbs while VLC is still running.
3) Start the channel interferer (i.e. another notebook running DL on the same channel) with BW_{max} of 10 Mbps while both applications are running.
4) Enable the optimizer.
5) Remove the channel interferer.

As it can be seen in Fig. 5 U_{node} drops slightly when we run both applications at the same time. The user can see some glitches on his movie. Then, a channel interferer is turned on and the utility values of both applications decrease drastically. In particular, VLC noticeably suffers from significant delay between video frames as depicted in Fig. 6. Also, DL experiences high throughput drop (see Fig. 7). After enabling the optimizer it notices this degradation from U_{node}. It then tries to see if there is a solution using optimization at the link layer or network layer. If it finds out that the other channels are so congested or reserved by the policy server which is the case here, it will try to tune application parameters according to their priorities. In particular, high priority applications like video streaming will get enough bandwidth while low priority applications have to sacrifice their bandwidth utilization. This is reflected in the node utility computation, in which the high priority application has a larger weighting coefficient than that of low priority applications. Specifically, The optimizer gradually tunes down BW_{max} of DL, resulting in lower delay in VLC and therefore higher flow utility value. It should be noted that the utility value of DL increases although BW_{max} is lowered. It is due to the fact that the utility value is realized from the optimizer perspective instead of user's perspective. In other words, the optimizer tries to maximize the network utilization by giving

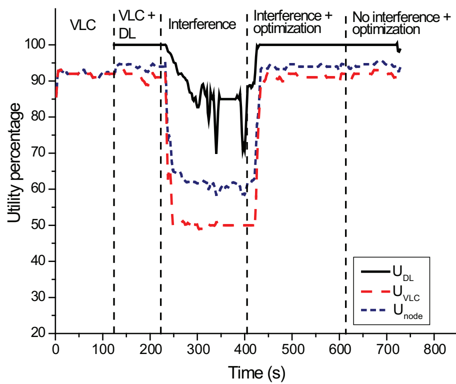

Fig. 5. The evolution of the utility values over time.

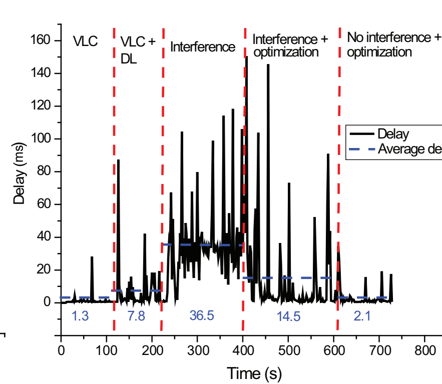

Fig. 6. The evolution of delay over time.

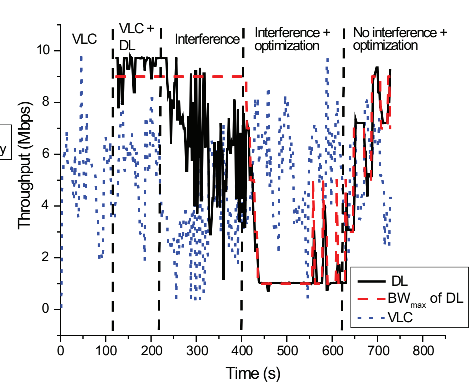

Fig. 7. The evolution of throughput over time.

more resources to high priority applications. Thereafter, some tuning steps have been performed. The video streaming again runs smoothly as shown in Fig.5. During the time between 550 and 620 seconds, we can observe several pulses from the DL application. This is due to the fact that after a certain time the watchdog timer is fired and then triggers the optimizer. Hence, the optimizer rapidly tries to increase BW_{max} of DL. Nevertheless, during the presence of interference, this change makes a sudden drop of utility value. As a result, the optimizer has to quickly tune down BW_{max} of DL. Finally, we remove the channel interferer. The optimizer sequentially increases BW_{max} and checks the resulting utility in order to improve the overall system performance.

VI. Conclusions and Future work

In this paper we have presented the design and implementation of utility-based optimization approach for wireless communication using CAPRI. The proposed system allows applications to express their requirements by registering their utility functions to a radio resource controller at run-time and in automatic way. In addition, we have shown that the system can potentially be employed to tune the application parameters in order to maximize network utilization. This is crucial especially in some circumstances when optimizer is limited with physical conditions where typical optimization cannot be used.

We have implemented a prototype for Windows/Visual C++. The utility evaluation duration tests show that introducing the CAPRI framework does not add considerable overhead and that the use of CAPRI framework for applications with soft real-time requirements is reasonable. The memory footprint tests also indicate that the CAPRI framework can be used on a notebook with limited memory resources. Additionally, we have addressed multi-application utility-based optimization through a practical case study. The results have shown that the proposed system maximizes network utilization and allocates the network resources according to the application priorities.

Our future work aims at performing a cross-layer optimization strategy that jointly optimizes the application layer, network layer, data link layer, and physical layer of the protocol stack using the proposed utility-based optimization algorithm. The implementation of CAPRI framework for Microsoft Visual C++ is available from [15].

Acknowledgment

The authors would like to thank the RWTH Aachen University and the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) for providing financial support through the UMIC research center. We would also like to thank the European Union for providing partial funding of this work through the ARAGORN and FARAMIR project.

References

[1] P. Mähönen, M. Petrova, J. Riihijärvi, and M. Wellens, "Cognitive wireless networks: your network just became a teenager," in *Proceedings of the 25th Conference on Computer Communications*, 2006, pp. 23–29.
[2] M. Petrova and P. Mähönen, *Cognitive Resource Manager: A cross-layer architecture for implementing Cognitive Radio Networks*. Cognitive Wireless Networks (eds. Fittzek, F. and Katz, M.), Springer, 2007.
[3] *EC FP7-216856 ARAGORN Project, Deliverable D2.4: Final System Architecture*, http://www.ict-aragorn.eu, last visited 04/04/2011.
[4] M. Sooriyabandara *et al.*, "Unified Link Layer API: A generic and open API to manage wireless media access," *Computer Communications*, vol. 31, no. 5, 2008.
[5] J. Riihijärvi, M. Petrova, and P. Mähönen, "A Common Application Requirement Interface for Cognitive Wireless Networks," in *in Proc. of 4th IEEE Workshop on Networking Technologies for SDR Networks in conjunction with SECON 2009*, Rome, Italy, 2009.
[6] J. Riihijärvi, M. Petrova, V. Atanasovski, and L. Gavrilovska, "Extending Policy Languages with Utility and Prioritization Knowledge: The CAPRI Approach," in *Proc. of IEEE DySPAN*, 2010.
[7] S. Khan, Y. Peng, E. Steinbach, M. Sgroi, and W. Kellerer, "Application-driven cross-layer optimization for video streaming over wireless networks," in *IEEE Communications Magazine*, vol. 44, no. 1, Jan. 2006.
[8] M. Ibnkahla, *Adaptation and Cross Layer Design in Wireless Networks*. CRC Press, 2008.
[9] C. Fonseca and P. Fleming, "Multi-objective optimization and multiple constraint handling with evolutionary algorithms - I: A unified formulation," *IEEE Transactions on Systems, Man and Cybernetics*, 1998.
[10] *Adobe Flash Player*, http://www.adobe.com/products/flashplayer, last visited 04/04/2011.
[11] *IIS Smooth Streaming for Microsoft Silverlight*, http://www.iis.net/download/smoothstreaming, last visited 04/04/2011.
[12] J. Levine, T. Mason, and D. Brown, "lex&yacc." O'Reilly Media, 1995.
[13] *Video LAN project*, http://www.videolan.org/vlc, last visited 04/04/2011.
[14] C. Boutremans and J.-Y. Le Boudec, "Adaptive delay aware error control for internet telephony," in *Proc. of 2nd IP-Telephony Workshop*, 2001, pp. 81–92.
[15] *ARAGORN project*, http://www.ict-aragorn.eu, last visited 04/04/2011.