DOI:
10.1007/978-3-319-59930-4_13

Document Version
Peer reviewed version

Multi-Agent System for Distributed Cache Maintenance

Santhilata Kuppili Venkata[1], Katarzyna Musial[2], Samhar Mahmoud[1] and Jeroen Keppens[1]

[1] Department of Informatics, King's College London, London, UK
{santhilata.kuppili_venkata}@kcl.ac.uk
[2] Faculty of Science and Technology, Bournemouth University, Poole, UK

Abstract. With the growing number of applications that require large data transfers from distributed databases, there is a great need for efficient distributed data caching methods. It is essential that data is cached at the best and optimal locations between users and data stores. Cache management should consider patterns about data usage and make dynamic decisions to place data across cache units. In this paper, we have modelled the distributed data caching mechanism using multi-agent system allowing to test strategies and algorithms for data placement that later can be incorporated in the real life applications. Subsequently, we demonstrate the application of this system to study various distributed coordination strategies for identifying effective data placement and thus improving overall cache performance. This study is significant for distributed system applications.

Keywords: Distributed cache, agent based modelling, coordination strategies

1 Introduction

Introducing multi-agent systems (MAS) into distributed computing can facilitate implementation and also provide novel characteristics such as more autonomy to the application system [22]. MAS allows construction of models to solve problems with variety of frameworks for environment centered analysis, design [3] and programmable architectures [9]. These architectures enable to create application examples such as distributed situation assessment, distributed coordination etc. to accurately represent and help researchers to develop new insights. Other examples include, large-scale distributed multi-agent systems in open systems such as E-Commerce [7], E-Health [14] and E-Governance [23]. Very few systems in distributed caching have implemented the agent-based approach. In industrial applications, TIBCO[3]. has come up with distributed cache scheme for distributed object management using MAS. In their work, MAS is used to define functions such as partitioning, replication, distribution, failure recovery and event handling. In another work in distributed caching, Dimakopoulos et al. [5] simulate peer-to-peer resource discovery using MAS. Each cache agent is used to store information to enable the distribution of data.

Distributed data caching is used in applications that need to cope with large volumes of data which are distributed all over the world[4]. For users' queries, data may have to be

[3] https://docs.tibco.com/pub/businessevents-express/5.2.1/doc/html/GUID-5CA44A37-01E9-4EE4-9922-8F8E70D50E7B.html

[4] www.ivoa.net

collected from multiple data stores before the reply is sent to the user. When groups of users work on related projects, queries tend to be repeated fully or partially. Repeated queries need same data to be retrieved and processed several times causing repeated data transfers, high bandwidth utilization and thus delayed responses [20]. Setting up several interconnected cache units to store the most repeated data at locations between users and data servers help to reduce response time and save processing resources [18]. Thus distributed caching is an interface between users and data stores.

Distributed caching is a complex system consisting of physical components such as multiple units of data servers, communication networks, middleware cache storage units, cache server (processing resources), and users. Cache management or maintenance is a software component which is considered to be the soul of the entire system. Maintenance typically happens on cache servers. Traditionally, cache storage units are small in size. Hence during the cache maintenance process, the decision has to be made about storing in cache units the most relevant data and removing the obsolete data. This means that we have to identify **'what data'** to store, **'where'** a given data segment should be stored, and for **'how long'**. This is the data placement problem in distributed cache maintenance. Periodically, an analyzer component (please refer to section 2) collects meta-data by performing an assessment of the data freshness and location relevance for each of the data segments stored. Analyzer helps cache maintenance to predict future needs based on the meta-data collected. In order to maximize cache utilization, management must employ approaches to make optimal decisions. Usually cache units are considered to be passive resource units and they are used only for storage purposes. But often global decision makers are hampered with knowledge about association between data units at a particular location. Also, as the overall system grows, global decision making may prove to be a bottle neck. To overcome these issues, we introduce the idea of delegating some responsibility to cache. With the knowledge about local data, caches actively participate in data placement decisions.

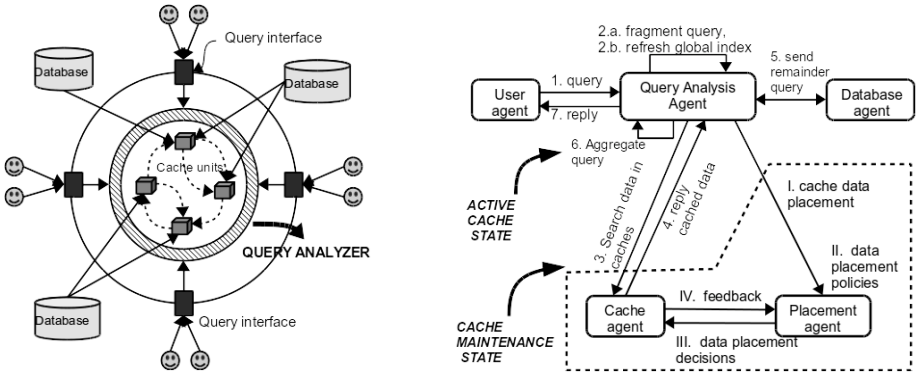
Typical applications that use distributed caches have huge number of cache units set worldwide. Coordinating management component, cache units should be able to analyze meta-data characteristics of the data usage and communicate with each other. All these entities are autonomous, intelligent, and contribute their knowledge towards solving data placement problem. We need to model interactions between these entities that cooperate and negotiate to make a collective decision about the best possible location for each data segment. All of these characteristics make agent-based system very well suited as a tool to model distributed caching and its processes. Therefore, we propose an agent-based design and agent-based simulation for evaluation of the presented ideas.

2 Background

Depending on application requirements, several types of architectures are available to describe the distributed cache system. The architecture we follow is as shown in Fig 1a. For the sake of clarity, we mention a data unit stored in cache as *'data segment'* and each cache storage unit as *'cache unit'* here after.

Each cache unit in the overall cache system stores data segments. A cache system can be in two states - (i) active state and (ii) maintenance state. Periodically, cache alters

between these two states. Usually, maintenance state is much shorter than active state. During the active state, the *query analyzer* receives requests from users and identifies part of the query that can be answered from the cache. It fragments the request and searches for the data needed by each of those fragments in cache units. For any data segment that is not found in a cache, the coordinator sends requests to databases. It then aggregates all segments together and sends is to user [11]. During this period, it collects meta-information about the user query patterns in order to predict future data needs.



(a) Distributed cache architecture (b) Multi-agent architecture for distributed cache

Fig. 1. Distributed cache system and multi-agent model

During the maintenance state, cache refreshment and data placement is performed. While coordinator keeps track of the changes in user query patterns globally, each cache unit governs the data segments stored locally. In smaller systems, the query analyzer can keep track of the global index of the data and hence user interests. But, when the system grows, some of the information is delegated to caches. Cache units keep track of the information related to each data segment stored at its own location. Hence, it is important to place each data segment at appropriate cache unit, so the overall performance of the cache system is maximised. Query analyzer and cache units should work together and coordinate their actions to maintain the overall cache system (shown in Fig 1b).

Typical diagnostics used for decision making in placing data segments are: *frequency* of each data segment queried, *time* when a data segment was used, *location* preference where the data segment was requested, *association* among data segments at a given location, *number of joins* in a query, storage *capacity* of the cache unit, and *workload characteristics* depicting the pattern of query requests.

Many researchers have worked in the area of distributed caching [21]. But since we are concentrated on semantic caching based on materialized views (a hybrid concept) in cooperative environment, we relate our work to this type of caching only. In an environment and goal similar to us, D'Orazio et al. [6] proposed a flexible locality based resolution and dual cache solution, based on semantic caching to improve query evaluation in grid middleware. But, their work does not use active cache participation. This solution may not be scalable due to the heavy cache operations. Lillis et al. [13] devel-

oped a cooperative caching scheme for XML documents. This scheme allows sharing cache content among a number of peers. The proactive cache replacement policy is implemented by each peer cache checking its nodes before performing a split whenever a specific node overflows. This work is similar to us but, since caches take decisions independently, they tend to miss global data access patterns. Our solution differs in this aspect. Cache units consult global information and other important diagnostics before taking decisions on eviction (explained later).

3 System Overview

3.1 Architecture

We have developed a multi-agent model for the distributed cache system. This model supports two main functions of distributed cache: (i) participation of agents in active state (regular query process) and (ii) cache maintenance for data placement (shown inside dotted lines of Fig 1b) in cache maintenance state. The system architecture together with major participating agents and their interactions is shown in Fig 1b. Identification of agents and their roles are modelled based on our earlier work [10]. We follow a flexible, generic MAS architecture that can use decision making and information gathering techniques. We have applied GAIA agent-oriented software engineering methodology [24] because of its capacity to formally describe agents in distributed systems. The functionality of agents and GAIA role models are presented in Table 1. Interaction diagrams to represent interactions among agents are developed using the standards defined for Agent Unified Modelling Language (AUML) [17].

User agents (UA) are modelled as the software representation of humans that query databases. Query process is instigated when UA sends a query to databases. Query response time is measured as the time elapsed from the query sent from UA to the reply received by a user (Fig 1b). The main responsibility of a user agent is to monitor the query response time. UA synchronizes its clock with the global clock to measure response time. During the query process, UA can be in one of the three states, *query sent*, *wait for response* or *query completion*. Also, user agents exhibit querying patterns related to their interests.

Query analysis agent (QAA) assumes coordinator role in the distributed caching. It has combined responsibilities for analysis and management. Hence QAA is a high level abstraction for multiple supporting agents. This agent assumes coordination and monitoring of the whole query-reply process. It interacts with UAs, maintenance agents and cache agents. In the active state, QAA is the single point access to user agents. It then fragments incoming queries, and searches within the cache for the data need by query. QAA divides query into fragments and resolves which part of the query can be answered by cache. It then sends the *remainder query* (part that cannot be answered by cache) to respective databases. After collecting all the data from sources, data is aggregated to formulate a response. QAA maintains the global index of data availability for lookup. QAA also gathers meta characteristics of user query patterns from the workloads during the active state. It sets diagnostics for the use during maintenance state. During the maintenance state, QAA runs prediction algorithms for future needs

with the help of diagnostics collected during the active state. With the help of other supporting agents QAA creates optimal data placement plans.

Cache agents (CA) are designed to take active part in cache maintenance. They are cooperative agents. Cache agents handle local data during active phase and prepare meta data to be used during maintenance phase. Meta data include knowledge about query pattern, data requirements and associations among data stored within a cache storage unit. CAs share information and negotiate with other agents while creating plans for ideal data placement. Cache agents are functional elements in deciding the scalability of the system.

Placement agent (PA) is an executor agent in the cache maintenance phase. It revises and recreates data placement plans and supports QAA during the maintenance state. PA interacts with cache agents to get feedback over the local information. PA holds multiple responsibilities. PA helps cache agents in negotiations. It aggregates plans made by cache agents and sends positive or negative feedback.

Database agents (DBA) are resource (passive) agents. They understand database load characteristics of the data usage and periodically submits this information to QAA. Database agents are mainly needed in the evaluation of database performance for various cache algorithms. DBA is responsible for assessing data store performance with respect to cache algorithms and decisions on replication.

Apart from the above main agents, **Negotiator Agent** supports QAA in handling negotiations among CAs. Similarly, a **Planning Agent** is another supporting role for QAA. Planning Agent is responsible for creating a master placement plan (distributed query planner) and distributing sub plans to others. **Communication Agent**, **Network Agent,** and **Processing Agents** have specific tasks in the overall distributed cache scenario, but they are not discussed in detail due to lack of space.

3.2 Coordination Strategies in Multi-Agent Systems

Many coordination strategies are available, each of them has its advantages and disadvantages and there is no universally best method [12]. We choose the most common strategies used in distributed computing [4] and multi-agent systems.

One of the foremost coordination approaches is the **master/slave** or **client-server** technique [16]. In this technique, the master agent plans and distributes fragments of plans to slaves. Master has the authority to do task and resource allocation. Slaves typically are cooperative in achieving common goals visualized by the master. Master/slave coordination approach is more suitable for centralized market structure. **Voting methods** [1] refer to techniques used to describe decision making processes involving multiple agents. Voting methods are useful in applications related to political science, game theory (for conflict resolution) and pattern recognition. In *weighted voting methods*, each vote carries equal weight while, *ranked* and *confidence voting methods* provide a bias to candidates. In **multi-agent planning** [16], agents build a plan that details all future actions and interactions required to achieve their goals as well as interleave execution with more planning and re-planning to avoid inconsistent and conflicting actions. In multi-agent planning, there is usually a coordinating agent that, on receipt of all partial or local plans from individual agents, analyses them in order to identify potential inconsistencies and conflicting interactions. The coordinating agent then attempts to

Table 1. Description of GAIA Role Model of Agents

The User Agent Role Model
Role Schema: User Agent (Software representation of a single or group of users).
Description: Agent is the instigator of query process. It calculates query response time
Protocols and activities: formulateQuery, sendQuery, set_LocalClock, receiveReply, calculate_responseTime
Permissions: prepares a Query, suspends queryState, reads queryStatus, accesses Globalclock
Responsibilities *liveness:* USER AGENT =(*formulateQuery.sendQuery*) (*receiveReply.calculateResponseTime*) (*setLocalClock*)
The Query analysis Agent Role Model
Role Schema: Query Analysis Agent
Description: Plays coordinator role. Monitors overall execution during active and maintenance states of cache
Protocols and activities: queryFragmentation, globalIndexUpdate, aggregateResponse, collectMetaQualifiers, prepareDiagnostics, createDataPlacementPlans
Permissions: reads workLoadCharacteristics; updates globalQueryIndex, reads userData, reads acceptQuery, prepares WorkloadAnalysis
Responsibilities *liveness:* QUERYANALYZER =(*startQueryProcess. globalIndexUpdate. aggregateResponse*), (*prepareDiagnostics*); MAINTENANCE-MANAGER = (*createDataPlacementPlans*);
The Cache Agent Role Model
Role Schema: Cooperative Cache Agent.
Description: Plays active role in cache maintenance. Coordinates with QAA,PA and peers to prepare data placement plans.
Protocols and activities: analyzeLocalData,vote, negotiate, generateLocalPlan
Permissions: acceses LocalSiteInformation, reads/writes/modifies LocalPlan
Responsibilities *liveness:* CACHEAGENT = (*analyzeLocalData.vote——analyzeLocalData.generateLocalPlan*) INFORMATION-EXCHANGER = (*negotiate*)
The Placement Agent Role Model
Role Schema: Placement Agent.
Description: Supports QAA in creating optimal placement plans based on various strategies; helps cache agents
Protocols and activities: collectVotes, collectPlans,negotiatePlans, collectQualifierData, createPlacement
Permissions: generates *Plan*, distributes *FinalPlan*, gathers *DataAnalysis,localCacheInfo*
Responsibilities *liveness:* PLACEMENT-HANDLER = (*collectQualifierData*),(*collectVotes——generatePlans*), (*collectPlans——generatePlans*), (*negotiatePlans——generatePlans*), (*createPlacement*)
The Database Agent Role Model
Role Schema: Resource role
Description: Agent asses data store performance characteristics
Protocols and activities: receiveQuery, lookupData processQuery, synchronizeClock, sendData
Permissions: access DataServer, process Query
Responsibilities *liveness:* DATABASE-SERVER = (*receiveQuery. synchronizeClock. lookupData.processQuery.sendData*)

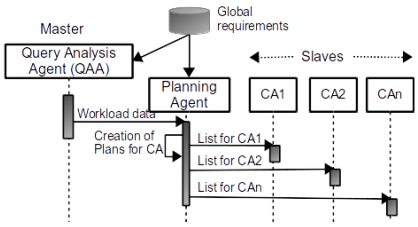
modify these partial plans and combines them into a multi-agent plan where conflicting interactions are eliminated. **Negotiation protocols** are used in the case where agents have different goals or the use of a resources by agents can prevent another agent to achieve its goal. The protocol followed in the negotiation and decision making process that determines each agent uses its positions and criteria for agreement [2, 15]. We also adopt a coordination approach from automatic control systems by obtaining **feedback** [8]. This strategy is similar to the effective negotiation, where agents reason their beliefs and desires [19].

3.3 Interaction Among Agents for Data Placement

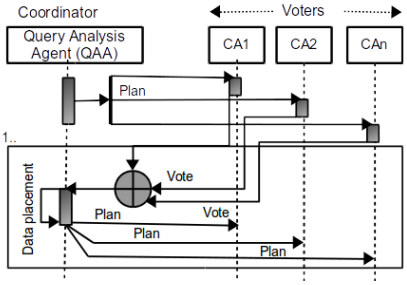
This section describes implementation of coordination strategies using the agent model. All strategies are assumed to follow standard rules: (i) all agents abide by the coordi-

nation by agreement (COA); as and when priorities and conditions of requirements change, coordinator agent broadcasts them to all participating agents; (ii) all agents accomplish coordination one phase at a time in a joint activity.

In **Master/slave coordination** strategy, query analysis agent (QAA) acts as the master coordinating agent as shown in Fig 2a. Master aims for equal distribution of data caching at each cache location. With the help of a planning agent, QAA decides the placement of data using first come first placed basis according to cache storage space availability. Thus master follows a greedy strategy and ensures to place each data segment at a first available best position. This strategy is the simplest of all and needs minimum number of inter agent-message communications. But, master/slave strategy suffers from improper distribution of data placement and thus longer query response time as there is no feedback from cache units (slaves).



(a) Master/slave strategy among cache agents and query analysis agent



(b) Voting strategy among cache agents and query analysis agent

Fig. 2. Master/slave and Voting coordination strategies in the system

Unlike master/slave, **voting strategy** enables cache agents to vote for the QAA's (coordinator) decisions. This strategy allows local interests of a cache to be expressed through voting as shown in Fig 2b.

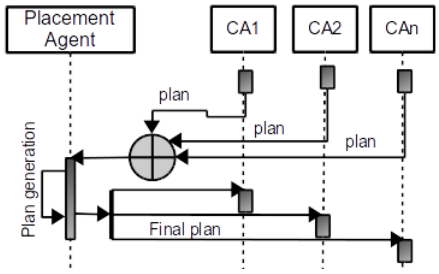
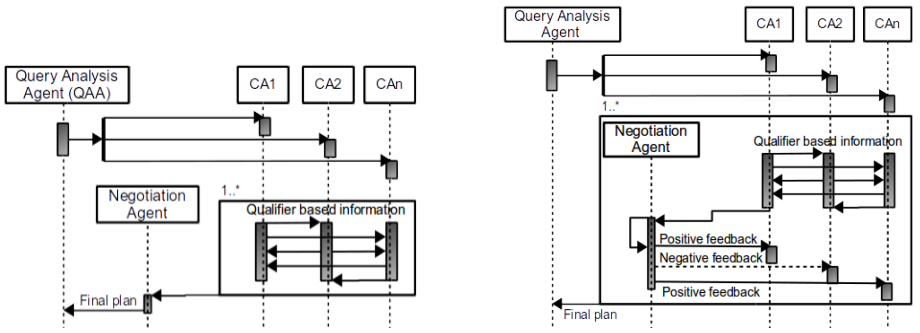


Fig. 3. Multi-agent planning

Cache units can vote based on the local knowledge (bias) such as affinity among all data stored within a cache unit. Polling of votes is done to accept or reject the whole plan. A plan is accepted only when it is accepted by majority of voters. Coordinator first starts with a basic plan. If rejected, improved plans are created by adding another qualifier to the heuristic. Coordinator follows a greedy strategy and ensures to place each data segment at first available best position. In **Multi-agent planning** strategy, cache agents develop plans keeping local benefit in view. Agents make individual plans using different heuristics. Here the Place-

ment Agent (PA) acts as coordinator and resolves conflicts and develops a new global plan. Coordinator resolves contention when more than one cache unit bids to store a specific data segment or placement of new data (shown in Fig 3). For example, a cache agent with larger data storage capacity may use storage capacity for heuristic where as another agent with high cache hit ratio might consider data frequency. PA must consider common interests to resolve conflicts. Thus placement agent follows a greedy strategy and ensures to place each data segment at a first best position.



(a) Negotiation among cache agents and query analysis agent in the system

(b) Negotiation with feedback among cache agents and query analysis agent

Fig. 4. Negotiation and Feedback coordination strategies in the system

In **Negotiation strategy**, cache agents negotiate with each other to maximize cache site utilization as shown in Fig 4a. In multi-agent planning, participating cache agents generate separate plans and submit them to the coordinator. Negotiation allows peer to peer communication with other cache agents to discuss plans. Negotiations are carried on till they reach to a mutually agreed solution. Each cache agent starts with their own objectives and benefits. This strategy uses all of its diagnostics to calculate the cost of placement to decide the ideal place. Hence many iterations of negotiations are needed before agents converge to a final decision. With a decentralized approach, the cache system may not suffer from bottlenecks with the scaling up of the system. Also, by considering multiple diagnostics, negotiation can predict user preferences well and recommend the most ideal place for each data segment. On the other hand, it suffers from the big inter agent message communication overhead. When negotiations run into infinite number of iterations, the coordinator agent (QAA in this case) may force cache agents to stop from going into infinite interactions.

Feedback strategy is an extension of negotiation strategy that aims to reduce inter agent message communication overhead. Feedback strategy employs a negotiation agent to provide feedback after every iteration to cache agents. It calculates the overall cost of data placement and provides feedback (shown in Fig 4b). When negotiations are not contributing to the improvement of the final results, negotiation agent may provide negative feedback refraining concerned agents from further negotiations. Thus feedback helps to reduce communication overhead and help the negotiations to converge quickly.

4 Evaluation

We have conducted a number of experiments to study various variables using Java based simulator developed for the research project. Due to space constraints studies related to three important metrics are presented. We have used synthetic workloads generated in our tool[5] to evaluate distributed strategies devoid of noise introduced due to communication networks, etc. Each workload is a set of queries with varied repetition distribution of queries. A workload is defined as a tuple: $\mathcal{W} =< \mathcal{N}, s, r, t, n >$; where, $\mathcal{W}$ is the workload, $\mathcal{N}$ = total number queries during the observation period, s = percentage number of queries repeated within the workload, r = statistical distribution with which s queries are repeated, t = statistical distribution with which queries are sent, n = number of cache agents in the experiment. For example, a workload <30000, 20, poisson, uniform, 45> describes a workload ($\mathcal{W}$) of 30000 queries; *20%* of queries are repeated in a *poisson* distribution among the workload; inter query arrival rate is set to *uniform* distribution; and number of cache agents = 45.

We made the following assumptions to maintain the uniformity across all strategies:

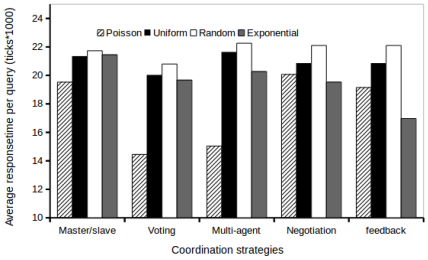
- All queries have equal complexity to keep the processing requirements equal.
- All cache units have identical server configuration. They are assumed to be located near to user groups. Hence cache agents can use location preference in their negotiations. Similarly, all data servers are assumed to have identical hardware configuration. We did not consider server-side cache for these experiments.
- Communication network is assumed to be congestion free and transmission lines are always available for data transfers. This assumption is valid to evaluate the performance of a strategy alone.

Average query response time is an important metric to evaluate cache performance. Response time is calculated as the total time elapsed between the time a query is sent from user agent to the time user agent receives response. Hence, response time depends on the data availability at a nearby cache location. Thus, response time indicates the effectiveness of a data placement as well. In a typical scenario several queries are sent simultaneously and the processing takes place in parallel. Here we calculated the average response time for a workload. Each of the experiments were repeated 8 times and median value is calculated below for the comparison study. Time spent for a process to complete is measured in terms of simulated time *ticks*. A tick is a unit time needed to complete it's execution.

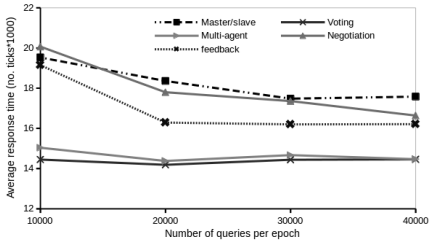
$$\text{Response time } = \frac{1}{N}\sum_{i=1}^{n}(\mathcal{D}_i + l_i + d_i + q_{proc}), \tag{1}$$

where, i = i^{th} query, $\mathcal{N}$= total number queries during the observation period, $\mathcal{D}$ = average processing time at data servers, l = cache latency (time spent at query optimizer + lookup time), d = data transfer time on network and q_{proc} = assemble time of cached data segments and remainder queries.

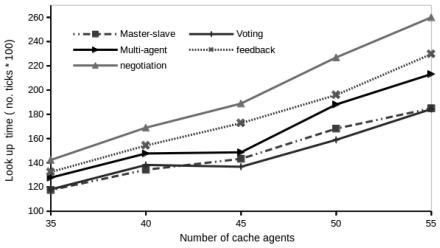
[5] Links to our query generator will be made public later.



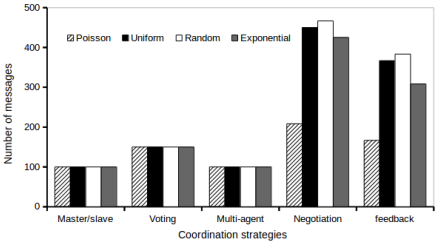
(a) Average response time with workload variation



(b) Average response time for poisson workload pattern



(c) Look up time variation



(d) Communication overhead with varying workload patterns

Fig. 5. Experimental evaluation

Average response time for varied query repetition distributions: Average response time was observed in this experiment for varied query repetition distributions as shown in Fig 5a. For workload $\mathcal{W}_1$ =< 30000,20,*,uniform,50>, each experiment was conducted several times and average was taken. We followed Least Recently Used (LRU) policy to for cache refresh during maintenance. From the results, random and uniform distributions of repetition of queries in the workload (where any particular query repetition pattern is not present) have resulted in the two highest response times across all strategies. This may be due to the deletion of queries based on LRU. Among the strategies, as master/slave does not consider cache agents' preferences, average time for master/slave has the highest response time over every query repetition pattern. Negotiation has exhibited high response times with large communication overhead. While poisson distribution has low response time consistently.

Average response time for varied number of queries: Based on the lower response time for poisson query repetition pattern as shown in Fig 5a, we focused on the response time with respect to increasing number of queries in Fig 5b. With workload $\mathcal{W}_2$ =<*,20,poisson, uniform,50>, almost all strategies stabilize with the increase in number of queries due to the heavy repetition of few number of queries in poisson pattern. Multi-agent and voting have low response times. Negotiation and feedback resulted in high response time and almost similar to Master/slave. But in general they are low as these strategies could find an ideal data placement better than other query

repetition distributions. Feedback has shown clear advantage over negotiation. This experiment is to test the scalability of coordination strategies for increased workloads.

Lookup time is another important metric for cache performance and a measure to consider for scaling up of the system. Lookup time is the time needed for query analysis agent to update query index and search for a stored segment due to reordering of data placement after each cache maintenance period. Fig 5c shows the lookup time needed for varying number of cache agents in the system for $\mathcal{W}_3 =<$ 30000,20,poisson, uniform, * $>$. Almost all strategies have linearly increased with increasing number of cache agents. Lookup time for negotiation and feedback are higher than others. This may be due to the higher number of data replacements done by them. Master/slave and voting strategies are quicker in comparison with other strategies and can help to scale the cache system.

The **communication overhead** in terms of number of internal messages needed for a strategy to reach a decision with workload pattern variation is shown in Fig 5d for $\mathcal{W}_4 =<$ 30000,20, *, uniform,50$>$. Master/slave, voting and multi-agent have the lowest overhead with finite number of communications per cache agent. These strategies are ideal for open systems that uses Internet as applications need to set up huge number of proxy caches over the network. Being iterative, negotiation strategy needed the highest number of internal messages. Though feedback is lower than the negotiation, the worst case for feedback may go up to the maximum similar negotiation.

5 Conclusion & Future work

In this paper, we have presented a multi-agent system to model distributed cache system and the study of optimal data placement for cached data to achieve higher performance. We chose master/slave, voting and multi-agent planning strategies to represent centralized coordination as well as negotiation to represent decentralized or peer to peer coordination in our study. We introduced a new feedback strategy to refine negotiation. Feedback will help to reduce the message explosion due to inter-agent communications in negotiation. Though master/slave is simple to implement, it has high response time due to the lack of knowledge about user preferences. In negotiation strategy the advantage of considering multiple diagnostics for recommending an ideal place is totally eclipsed by the inter-agent communication overhead. Limitations on the evaluation is not extensively discussed as the main aim of this paper is to present the MAS. In future, we would like to implement other coordination strategies with cache refresh policies. We also plan to incorporate this model in real life applications and compare with existing non multi-agent approaches.

References

1. T. Bosse, M. Hoogendoorn, and J. Treur. Automated Evaluation of Coordination Approaches. In P. Ciancarini and H. Wiklicky, editors, *Coordination Models and Languages*, volume 4038 of *LNCS*. Springer Verlag, 2006.
2. S. Bussmann and J. Müller. A Negotiation Framework for Cooperating Agents. In S.M.Deen, editor, *Proc. of the CKBS-SIG (CKBS'92)*, 1992.

3. A. Consoli, J. Tweedale, and L. Jain. An Architecture for Agent Coordination and Cooperation. In *Knowledge-Based Intelligent Information and Engineering Systems*, volume 4694 of *LNCS*. Springer Berlin Heidelberg, 2007.
4. G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems - Concepts and Design*. Addison Wesley Publ. Comp., 5 edition, 2011.
5. V. V. Dimakopoulos and E. Pitoura. *A Peer-to-Peer Approach to Resource Discovery in Multi-agent Systems*, volume 2782 of *LNCS*. Springer Berlin Heidelberg, 2003.
6. L. d'Orazio, F. Jouanot, Y. Denneulin, C. Labbé, C. Roncancio, and O. Valentin. Distributed Semantic Caching in Grid Middleware. In *Database and Expert Systems Applications, 18th Intl Conf, DEXA Proceedings*, 2007.
7. M. He, N. R. Jennings, and H.-F. Leung. On agent-mediated electronic commerce. *IEEE Trans. Knowl. Data Eng.*, 15(4), 2003.
8. J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
9. K. Kravari and N. Bassiliades. A Survey of Agent Platforms . *Journal of Artificial Societies and Social Simulation*, 18, 2015.
10. S. Kuppili Venkata, J. Keppens, and K. Musial. Agent Based Simulation to Evaluate Adaptive Caching in Distributed Databases. In M. Rovatsos, G. A. Vouros, and V. Julián, editors, *EUMAS/AT*, volume 9571 of *LNCS*. Springer, 2015.
11. S. Kuppili Venkata, J. Keppens, and K. Musial. Adaptive Caching Using Sub-query Fragmentation for Reduction in Data Transfers from Distributed Databases. In N. P. F. Lorente and K. Shortridge, editors, *ADASS XXV*, ASP Conf, Ser. ASP, 2016.
12. V. Lesser and D. Corkill. Challenges for Multi-agent Coordination Theory Based on Empirical Observations. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, 2014.
13. K. Lillis and E. Pitoura. Cooperative XPath Caching. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08. ACM, 2008.
14. S. Mahmoud, G. Tyson, S. Miles, A. Taweel, T. V. Staa, M. Luck, and B. Delaney. Multi-agent system for recruiting patients for clinical trials. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14*, 2014.
15. B. Marzougui and K. Barkaoui. Interaction Protocols in Multi - Agent System s based on A gent Petri Nets Model. *(IJACSA)*, 4(7), 2013.
16. H. S. Nwana, L. C. Lee, and N. R. Jennings. Co-ordination in Software Agent Systems. *The British Telecom Technical Journal*, 14(4):79–88, 1996.
17. J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. *Ann Arbor*, 2000.
18. T. M. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.
19. K. P. Sycara. Multiagent Compromise via Negotiation. In M. Huhns, editor, *Distributed Artificial Intelligence (Vol. 2)*. Morgan Kaufmann Publishers Inc., 1989.
20. A. S. Szalay, J. Gray, A. R. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg. The SDSS Skyserver: Public Access to the Sloan Digital Sky Server Data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02. ACM, 2002.
21. T. T. Team. Mid-tier Caching: The TimesTen Approach. In *Proceedings of the ACM SIGMOD Intl Conf on Management of Data*, SIGMOD '02. ACM, 2002.
22. M. Štula, D. Stipaničev, and L. Šerić. *Multi-Agent Systems in Distributed Computation*, volume 7327 of *LNCS*. Springer-Verlag Berlin Heidelberg, 2012.
23. M. Warnier, F. M. T. Brazier, and A. Oskamp. Security of distributed digital criminal dossiers. *Journal of Software*, 3(3), Mar. 2008.
24. M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.