CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2009)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1444

Economic aspects of building software for service-oriented architectures

Dimitrios Antos*,†, Costas Courcoubetis
and George D. Stamoulis

Department of Informatics, Athens University of Economics and Business, Athens, Greece

SUMMARY

The concept of service-oriented architectures (SOA) has recently emerged as a design principle for the next generation of IT solutions. The main idea behind SOA lies in treating software applications as composed of simpler, cooperating services, implemented by components that communicate over networks through open standards. Such a modular structure is expected to question the effectiveness of current business models for building and distributing software, which generate revenue through license fees. Under SOA, the license fee is replaced by more flexible schemes, whereby users are charged according to their actual usage of services and the corresponding hardware infrastructure. Our work constitutes a first step toward exploring the economic aspects of a market for software components, as well as the incentives of software manufacturers to support this trend. In particular, we examine the factors that affect profitability in an open market for services, we build simple models to predict and explain market growth and we also suggest ways to accelerate this growth, while also achieving a higher level of economic efficiency. Copyright © 2009 John Wiley & Sons, Ltd.

Received 23 October 2007; Revised 27 February 2009; Accepted 10 March 2009

KEY WORDS: service-oriented architectures; Grid; economics

1. INTRODUCTION

Software developers have been investigating better and economically more efficient ways of building software applications for decades. Technological innovation and competitive pressures have systematically pushed for newer and smarter techniques to be formulated and employed, gradually moving the community of software developers away from monolithic practices and closer to

*Correspondence to: Dimitrios Antos, Department of Informatics, Athens University of Economics and Business, Athens, Greece.
†E-mail: dantos05@aueb.gr, antos@fas.harvard.edu

Contract/grant sponsor: EU-funded GridEcon project; contract/grant number: FP6-2005-IST5-033634

modular approaches. Among the highlights of this trend were the introduction of the object-oriented programming languages in the 1970s, the emergence of independent software vendors a few years later (who specialized in developing reusable components for other software manufacturers), the appearance of frameworks (large collections of general-purpose reusable libraries targeted at supporting and accelerating the development of various applications—examples include Sun's J2EE and Microsoft's .NET Framework) and recently the definition of protocols capable of providing remote code execution and web services.

Today, this trend is expected to continue with the emergence of pervasive computing systems [1] and service-oriented architectures (SOA). The former term refers to the existence of computational capabilities within virtually all objects of human surroundings and not just computers or cell phones. This scenario portrays desks, furniture, electric appliances, cars and even our pens and clothes as being connected to the Internet, communicating with each other and offering all kinds of computational services to human users. Such a highly distributed context will eventually lead to SOA, whereby the computational needs of individuals and corporations will be served by combining independent, cooperating web service-like modules over communication networks. Some also envision a new generation of Internet applications, structured as components located in various locations and communicating among themselves to provide comprehensive services. A setting like this is close to utility computing [2], a paradigm in which information technology is provided as a commodity (e.g. like electricity) and its users are charged on a per-use basis, while being ignorant of the actual location of the computing facilities and the network technologies employed to distribute the service.

SOA clearly exhibit a radically different set of characteristics, compared with more traditional approaches of developing and selling computer software. Before SOA, the dominant business model was that of building applications as packages and charging their users a (usually expensive) license fee. Variations like group licensing are just price-discrimination strategies of this fundamental model. Software leasing is another variant, whereby the customer does not have to pay for the license immediately, but in small increments while using the product. Application service providers (ASPs) are perhaps an intermediate step toward SOA. In this model the customer is charged a license (or per-use), but the software is installed in the ASP's computing facilities. Thus, the end-user does not have to worry about maintaining, protecting or operating private infrastructure, something that usually entails considerable costs. SOA, on the other hand, take this notion one step further, in that software need not necessarily be offered by a single provider, or installed in predetermined facilities. Independent, cooperating components may be offered by multiple providers, reused in a number of applications (besides the one they were initially developed for) and run over the facilities of an ASP or multiple infrastructure operators. This component reuse is a fundamental aspect of SOA and motivates our research in this paper. It allows for large cost reductions both on the provider and the customer side as discussed in [3]. The advantages of SOA and some interesting market forecasts can be found in [4,5]. There is an obvious strong complementarity between SOA and Grid computing as discussed in [6], and there are strong drivers for a SOA-cloud integration [7].

This study analyzes various aspects of SOA from an economic point of view. Traditional software development largely follows the economic patterns of low marginal cost, positive network externalities from the demand side and lock-in. However, the patterns of production, reuse and consumption under SOA are not studied in the current economic literature and are expected to raise a number of new challenges of a socio-economic nature. For example, it is not yet clear whether a market for components is feasible, i.e. whether manufacturers can sustain themselves. Does the survival of such a market require software developers to exhibit a large degree of cooperation? Are production externalities present and how do they affect market evolution? Do first-movers enjoy a better chance of ruling the market or increasing their profitability? How can the various pricing strategies affect the behavior of the market as a whole? These, among others, are crucial questions that our work addresses.

Our goal is to understand the implications of component reuse on the supply side. As we will argue in the remainder of the paper, this cost reduction implied by component reuse does not come for free. When application providers plan the development of their applications, they face the added risk of not having certain required components available in the market, or even having to design an already available component again (and hence increasing their production costs). We also observe that the revenue of a provider that produces components is related to the number of other components available in the market due to the large number of possible combinations that can be created as applications for the customers. This is an important externality that has important consequences for the dynamics of the market. We believe that our modeling of the SOA producer incentives is novel and useful for understanding the certain aspects of this new technology. It raises a large number of research topics that should be investigated further. Our results in this paper do not offer direct advice to practitioners but help in understanding some key trends in this new market. Our discussion on pricing (either through a common repository policy or in a free market) shows that prices may play a very important role in the way, the market will develop. Of course, all this is sensitive to the assumptions of our models and to the values of certain parameters.

The structure of this paper is as follows. Section 2 places our work within the existing literature. Section 3 examines the decision context of two software developers who wish to build applications sharing some common functionality. This simple case serves to exemplify the risk associated with being a first-mover in a market of software components and illustrates the need for cooperation among entrants. Section 4 discusses the implications inherent in such a cooperative attempt and outlines some of the fundamental issues. Moving beyond one-shot cases, Section 5 examines the expected profitability of a participant in a market for components and how this is affected by the size of the market and the prices within it. Section 6 deals with the initial stages of market growth and suggests ways to accelerate it, with interesting implications to the economic efficiency of the market as a whole. Finally, Section 7 studies the pricing strategies that are likely to be employed by profit-maximizing participants and draws qualitative results as to how these will affect market evolution. The paper concludes with a short summary and a discussion on the directions for future work.

2. RELATED WORK

The most closely related work from the economics literature is the 'mix-and-match' approach, originated in [8], and followed in [9,10]. The basic idea of the mix-and-match models is the following: in industries where consumers can assemble their own systems, firms must decide whether to make their components compatible with those of their rivals, before competing with the prices. These models look at the case of two firms producing the two sole components of a system; firms are assumed to be profit maximizing. If the two firms choose incompatibility, then only two systems are available for the consumers to buy: firm A's system X_{AA}, which combines components X_{1A} and X_{2A}, and firm B's system X_{BB}, which combines components X_{1B} and X_{2B}. If the components are compatible then the consumers have also the options to form the systems X_{AB} and X_{BA}. There are variations of these models, in which compatibility can be decided unilaterally or must be decided by both firms. In this context they show that, under certain conditions (absence of positive externalities), the incentives of the firms are aligned. Other related work is in [11], which examines the impact of the number of varieties of complementary goods on the decisions of consumers to buy certain primary goods under conditions of incompatibility.

The above models have some common aspects with those defined in this paper, namely, that cutting the price of one component will increase the sales of all systems using that component, including systems produced by other firms. But there is an important difference: in our model a firm decides whether to produce a given component or not, which is not the case with the models in the mix-and-match systems literature, where all the components are always produced by each provider. An other important difference concerns the type of competition. In the mix-and-match systems, firms produce systems that are substitutes, and create competition. In our model, where systems correspond to complete applications, we assume that there is no substitution, i.e. each new application generates its own demand. The effect of compatibility in our model lies in the cost reduction, it enables for the firms that reuse the components developed by other firms. We assume that firms producing components are profit-maximizing, and they do so by choosing component prices, see Section 7. To our knowledge, this model is novel and better captures the aspects of the software industry and cloud computing that we are interested in. We also believe that the results of this paper cannot be derived by simply extending or applying existing results.

Compatibility has also been discussed in other contexts. It has been traditionally associated with network externalities, as in the standard framework first explored in [12]. A system of compatible components is treated as a single good characterized with positive consumption externalities since the utility of a consumer increases with the number of other consumers using compatible products. This is the standard model that justifies compatibility: a larger 'network' of compatible components may be more profitable since the demand will be higher. It suggests that small firms producing components will tend to choose compatibility, whereas larger firms that already have a large established market share may prefer to be incompatible with others. All these results assume externalities associated with demand.

In our present work our models also show the existence of externalities, but on the supply side. These are now between producers that benefit from the availability of compatible components provided by other producers, since this allows them to reduce their costs. In that respect, our models and the corresponding results are new and depend on the specific form of the externalities obtained due to cost reduction. For instance, the specific formulas of the provider's profit depend on the state of the system (see Sections 5 and 6) and the different results we summarize in Figure 1 depend on the above structure of the problem and are not generic. The specific structure of our model makes them plausible.

To complete our discussion, there are several related recent studies on complementary technologies and patents that investigate the issues of double markup (see [13,14], among others). In [13] the authors analyze the incentive of a monopolist in product A to enter complementary product B's market in order to force independent suppliers of B to charge lower prices, which increases its own profits made from product A. In our case this would model the competition between providers that

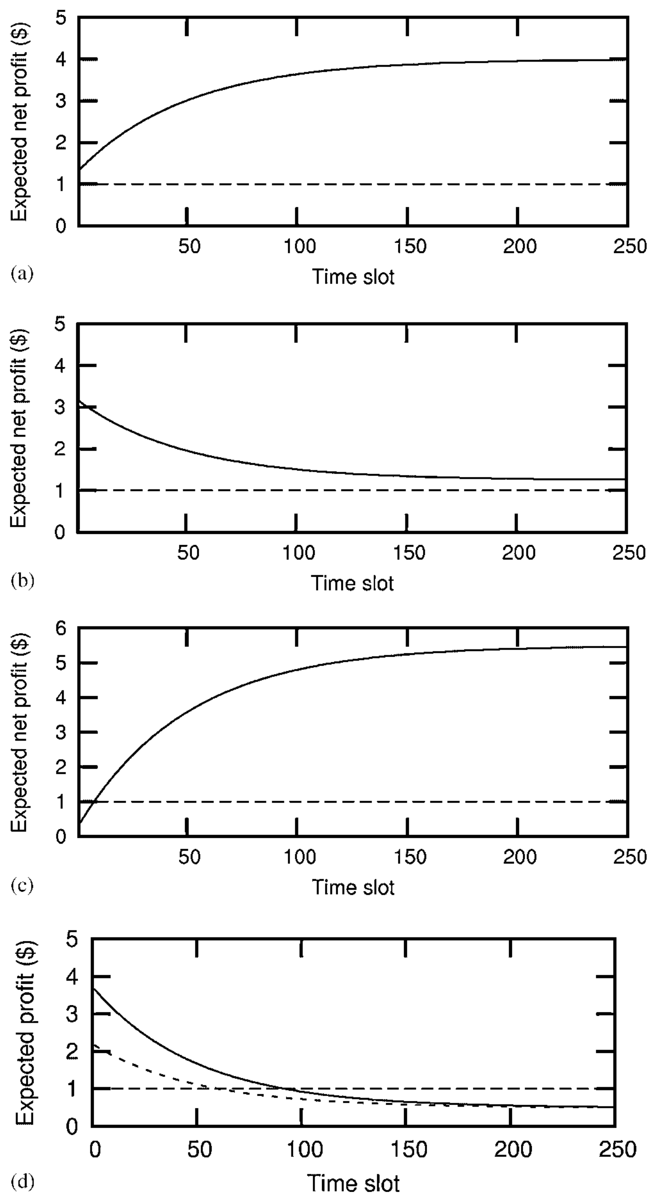

Figure 1. Developer profitability with time of entry.

reduces the incentives for double markup. An interesting model that has also some commonalities with our model is in [15,16]. This work deals with models of patent portfolios that allow a full range of complementarity and substitutability, and analyzes the welfare effect of patent pools and evaluates several factors that encourage or hinder the formation of these pools.

3. THE ONE-SHOT GAME

We begin our exploration by examining the decision-making problem of two software developers, A and B, each of whom desires to build and deliver an application to the market. Applications are made up of a number of *modules*, each of which represents a subset of the overall functionality within the application. Under the current business model for software delivery, both developers have to build all modules of their application and sell them as a *package* for a license fee. Assume that developer i (with $i \in \{A, B\}$) has cost c_i to build one module and earns revenue R_i by delivering its application at a profit-maximizing price. In addition, let for simplicity both applications be made up of k modules. Therefore, under the license scheme, application developers make profit $\pi_i^{pack} = R_i - kc_i$. Notice that applications do not compete with each other and are assumed to address different market needs, so that none of the developers gain anything by blocking the completion of someone else's application. We also consider risk neutral firms with quasi-linear utility functions that depend only on their revenue.

Under SOA, however, developers have other options as well. Instead of delivering their applications as packages, they may develop each module as a component with open interfaces. This does not affect their application's functionality, since components developed for an application may call each other's methods and interact in an open environment, such as a Grid system, with this being transparent in the eyes of the end-users. However, choosing to build components instead of a single large package brings about two significant changes: on the one hand, there is a chance that in the applications of A and B there will be common functionality, i.e. some of the modules will be used in both applications. Developing a common module as a component gives manufacturers the opportunity to build it only once and then share it, so that their joint development costs are reduced. On the other hand, though, a component requires open interfaces to communicate with the rest. Plus, breaking up a complex application into neat smaller components with well-defined semantics that can be generically used, which might not be a trivial task. Therefore, choosing to deliver components leads to increased development costs per module. To capture the above two effects, we assume that m of the k components of our developers' applications are shared, i.e. are common to both products, and that building a module as a component with open interfaces increases its production cost by a factor of $\lambda > 1$. We also assume that the number and identities of the m common components are common knowledge to both developers A and B, meaning that there is no uncertainty as to which subsets of functionality are in common with their applications.

If a developer chooses to deliver components, it must also decide which of the m common components to develop itself. Cost reductions are achieved when a common component is developed by just one of two participants and then shared with the other, at a price r that is negotiated between them. In actuality, developers under SOA, even with full knowledge of the commonly needed modules have an additional number of 2^m choices, since any of the m common components may or may not be built by a developer.

The above scenario defines a *game*, in which A's payoff is affected by B's decision and vice versa. Of course, if either developer chooses to deliver a package, it makes π_i^{pack}, irrespective of what the other developer selects. Both players (developers) in our game have $2^m + 1$ strategies to select from. Strategy P corresponds to delivering a package, while each other strategy C_X represents delivering components: building those which are unique to its own application and, for those in common, only the ones in the set X. Let $\pi_i^{S_i|S_j}$ be the profit of developer i when having selected strategy S_i and the other player $j \neq i$ has chosen strategy S_j. Naturally, $\pi_i^{P|S_j} = \pi_i^{pack}$ for all $S_j \in S$.

For all other cases (when $S_i \neq P$), the payoff is formulated as follows. Assume $S_i = C_X$. Since i is building components, it has to develop each of the non-common components $k - m$ at a cost λc_i per component. Moreover, it has to develop the common components in X for the same cost per component, leading to an additional cost of $|X|\lambda c_i$. As for revenues, manufacturer i is able to deliver its application when all of its components are available. Therefore, i makes R_i if the other developer has chosen C_Y and $X \cup Y = M$, where M is the set of all common components ($|M| = m$). In that case, however, developer i has to purchase the remaining components in $M - X$ from the other player, at a price r per component.

We remind the reader that, by assumption, developers are aware of which components they have in common but do not engage in negotiations to mutually decide, which components in M will be developed by each. Instead, we assume that they independently and simultaneously decide on a strategy for delivering their applications. To examine their behavior under this context, we employ the concept of a *Nash equilibrium* from game theory. A Nash equilibrium is a vector of strategies $(S_1, \dots, S_N)$ in a game with N players, where S_i is the strategy chosen by the ith player, such that no player has the incentive to deviate from its chosen strategy unilaterally. Formally, if S_{-i} is the strategy vector of all other players except i, and $\pi_i^{S_i|S_{-i}}$ is the payoff to player i when he selects S_i and everybody else chooses the strategies in S_{-i}, then, under a Nash equilibrium, $\pi_i^{S_i|S_{-i}} \geq \pi_i^{S'_i|S_{-i}}$ for all $S'_i \in S$.

To identify the Nash equilibria (NE) of our game, we reason as follows. First, the strategy profile (P, P) is always a Nash equilibrium. To see that, assume that A chooses P. Then, to have all the components for its application developed, B may either choose P or C_M, otherwise at least one component will be missing. Under C_M, however, B will make $R_B - k\lambda c_B$, which is less than $R_B - kc_B$ for all values of $\lambda > 1$. Therefore, B will choose P and thus (P, P) is a NE.

Moreover, every strategy pair (C_X, C_{M-X}) is also a NE, for price r and values of λ in an appropriate range. In each such case, A makes $R_A - (k - m + |X|)\lambda c - r(m - |X|) + r|X|$ and B makes $R_B - (k - |X|)\lambda c - r|X| + r(m - |X|)$. For this to be a Nash equilibrium, it must be that both these quantities are greater than or equal to the corresponding π_i^{pack}. Observe that, in the NE of this type, all modules are built as components and each is built only once (by only one developer).

It can easily be seen that, if (C_X, C_{M-X}) is a Nash equilibrium, it also represents a Pareto-improvement upon (P, P), meaning that both developers are better-off under (C_X, C_{M-X}) than under (P, P). This is obvious since both the developers' payoff is greater than π_i^{pack}. However, as we have assumed no coordination between developers (independent decisions) then both cannot be aware of the exact set X, the other player will choose to develop (so that they may develop $M - X$). Worse, if they independently choose X and Y and $X \cup Y \subset M$, then neither can deliver its application, both make zero revenues and thus incur losses. Finally, the NE of the form (C_X, C_{M-X}) are not symmetrical. But since neither firm can identify itself as player A or B, they cannot safely decide between strategies C_X and C_{M-X}. The only symmetrical NE in pure strategies are (P, P), while there clearly exists no dominant strategy for the players.

What about mixed strategies though? A *mixed strategy* for player i consists of a randomization over the strategies in S, whereby each strategy $s \in S$ is chosen with probability p_s and $\sum_{s \in S} p_s = 1$. Calculating the symmetrical mixed strategy Nash equilibrium gives non-zero probabilities for all

Table I. A simple game payoff matrix.

	P	$C_\emptyset$	$C_{\{\gamma\}}$
P	$R-2c$	$R-2c$	$R-2c$
	$R-2c$	$-\lambda c$	$R-2\lambda c$
$C_\emptyset$	$-\lambda c$	$-\lambda c$	$R-\lambda c-r$
	$R-2c$	$-\lambda c$	$R-2\lambda c+r$
$C_{\{\gamma\}}$	$R-2\lambda c$	$R-2\lambda c+r$	$R-2\lambda c$
	$R-2c$	$R-\lambda c-r$	$R-2\lambda c$

strategies in S. However, the expected payoff under this strategy profile is exactly equal to π_i^{pack} for each developer i. To see this, we employ Nash's fundamental theorem [17]; this states that, under a mixed strategy Nash equilibrium, the payoff of every pure strategy of player A played with non-zero probability against the randomized strategy of player B must be the same. Since pure strategy P has payoff π_A^{pack} whatever the probabilities in B's strategy, then every other strategy C_X must have payoff π_A^{pack} as well, for a total expected payoff of exactly π_A^{pack}. The same applies for player B.

The following payoff matrix exemplifies the simplest case for $k=2$, $m=1$ and identical players ($R_A=R_B=R$ and $c_A=c_B=c$). Developers A and B each have an application to build: A's application consists of modules α and γ while that of B is made up of β and γ. Strategies of player A are represented as rows and those of player B as columns. Each cell of the table contains the payoffs of A and B, in that order, given the intersecting rows and columns corresponding to their choices (Table I).

Here, we clearly see that $(C_\emptyset, C_{\{\gamma\}})$ and $(C_{\{\gamma\}}, C_\emptyset)$ are NE for sufficiently low λ and a suitable value for r. In particular, for $(C_\emptyset, C_{\{\gamma\}})$ to be a Nash equilibrium, it must be $R-\lambda c-r>R-2c$ and $R-2\lambda c+r>R-2c$. Both these conditions are true if and only if $\lambda\leq\frac{4}{3}$ and $r\in[2c(\lambda-1), (2-\lambda)c]$. In that case, these equilibria are better for both developers than the third NE, which is (P, P).

The intuition we gain from the above analysis is that building components can be profitable, yet, without coordination among developers, it can be a *risky* business. If a developer is not certain that it will indeed enjoy cost benefits due to common modules being developed by others, then it faces the risk of not 'completing' its application (i.e. having all of its modules). Alternatively, the developer faces the risk of having to incur the cost λ times for every module without any actual benefits. Furthermore, even if a company is willing to take a chance and try out components, the expected profit is no greater than that of a package. Plus, a package guarantees this profit, while anything else gives it as an 'expected' payoff. If we assume that players are even slightly risk-averse, i.e. they have sublinear utility functions, a developer is clearly better-off choosing to deliver a package.

A variation of the above game can be formulated, in which developers do not choose strategies simultaneously but in order. Suppose in our previous example that A plays first and B follows. Naturally, if A chooses to build a package, then B will do so as well, so (P, P) remains a Nash equilibrium. If A chooses strategy C_γ and selects price r for the common component γ, then B will select $C_\emptyset$ if $r<c(2-\lambda)$; otherwise, it will choose P. In the first case, the revenues of A increase with r, so A will rationally select the maximum possible price for r, conditional on B purchasing the common component from it, i.e. $r=c(2-\lambda)-\varepsilon$, for a very small $\varepsilon>0$. Finally, there is an equilibrium in which A chooses $C_\emptyset$ and B selects C_γ and price $r=c(2-\lambda)-\varepsilon$.

Observe that, in this variation of the game, the player who builds the common component makes $R-2\lambda c+c(2-\lambda)-\varepsilon \approx R-2\lambda c+2c-c\lambda = R-3\lambda c+2c = R-c(3\lambda-2) \geq R-2c$, if $\lambda \leq \frac{4}{3}$. We see that the developer of the common component makes more profit than the developer who purchases it and more than p_i^{pack}. Therefore, the first developer will optimally choose C_γ and B will select $C_\emptyset$. Notice that, in this equilibrium, the first player effectively absorbs all the benefit from delivering components of the second player.

Notice, however, that most hardships in our game stem from the fact that developers are not allowed to negotiate before making a decision. What if we allowed them to decide, *a priori*, which of the common components would be developed by whom, as well as the exchange price per component? We study this scenario in the next section.

4. ISSUES IN COOPERATION

In the previous section, we have formulated a simple game to assess the decision-making problem of developers who wish to achieve cost reductions by developing commonly used components in their applications. We have seen that, if firms do not negotiate on the exact subset of the common components that each will develop, they are very likely to totally refrain from delivering the components. Therefore, the element of cooperation becomes critical to the appeal of component development.

By cooperation, we mean that developers have a chance to decide, before any decision is made, which components will be developed by whom and at what price r will they be shared. In this cooperative scenario, there is no reason why only one-on-one interactions should occur. Instead, we can imagine larger groups (coalitions) of cooperating developers being formed, in which the costs of developing common functionality are shared. In this section, we are interested in identifying ways for developers to engage in such cooperative efforts. The analysis is thus aimed at exposing any implications, as well as providing simple solutions whenever possible, employing concepts from cooperative game theory.

4.1. Deciding roles

The first question that a set of cooperating developers has to answer is who builds what. First of all, it is reasonably clear that no developer will purchase a component at a price r that is greater than its actual development cost λc_i. We may now identify two subcases to study the above question, based on this fundamental observation and our assumption of the developers' rationality. In the first subcase, all developers are *identical* in terms of cost efficiency, i.e. $c_i = c$, for all i. Although not entirely accurate, this hypothesis is not outright false, since the development costs per component across the software industry are not expected to differ significantly. In the second subcase, developers are assumed to have different levels of cost efficiency, expressed as different c_i's.

Distinguishing between these two subcases helps expose some aspects of developer cooperation that are more (or less) important in each. For instance, if we assume identical costs, the question of which firm (A or B) develops a commonly used component affects just their profitability. On the contrary, if firms have varying costs, then a *social optimum* is obtained by having the most efficient developer actually build the component, in the sense that society pays the least possible amount of money to develop some reusable functionality. Furthermore, if the cost efficiency (the

c_i's) of participants in a coalition are *hidden information* and they have the option of misreporting their actual cost values, then there is a need for incentive-compatible mechanisms to guarantee the coalition's efficiency; that is, guarantee that, despite the information asymmetry, the most efficient developer still produces a commonly used component.

We begin with our first subcase, in which all firms are identical in terms of their costs (i.e. $c_i = c,\ \forall i$). Here, a direct result of the necessary condition that $r < \lambda c$ is that no developer will ever rationally build a component not required by its own application. To see this, consider the case of just two developers, A and B, as before. If A does not require a component for its application but knows that B does, he can build it at cost λc but sell it for $r < \lambda c$, in which case he experiences a loss. If now, instead of a single developer B, a group of developers (say, G) needs this component, developers in G are clearly better-off sharing the cost among themselves, rather than each paying r to A. For instance, let there be two developers in group G. For A to be profitable, it must be that $2r > \lambda c$, meaning that the joint revenue from both developers in G purchasing the component should outweigh its development cost. This implies that $r > \lambda c/2$. But, by appropriately coordinating, developers in G can easily choose one of themselves (at random) to develop the component for λc, then share it for half that cost, so that each will only have to pay $\lambda c/2$. The fact that this price can indeed be achieved is proved in Section 4.2. A similar argument applies for any number of developers in G.

We see that if developers have identical cost levels, then they must negotiate for every component within the group of participants that need this component, since no 'outsider' may affect their decision, by accepting to build the component and sell it to them (once to each member of the group). Within those groups, of course, there is no point for two (or more) firms offering the same component; not only would this be inefficient as a whole, but also irrational for the second developer. Therefore, the group must elect a single developer to do the job, then share it with others at a predefined price. Naturally, the group will elect the developer who promises to share it at the lowest price. It turns out that such a price is unique and we calculate it in Section 4.2.

Notice that, in the case of multiple common components, the decision as to which developer builds each of them can be handled separately. For each of the common components, a separate group is assembled to assign its development to a particular member. For example, if a developer belongs to a group G_α for the common component α, it may also belong to another group G_β for component β, etc.

Turning our attention to our second subcase, however, we see that an efficient developer may as well build a component not included in its application and sell it at a price r, which is higher than its own cost but lower than others', even if these form a coalition to share the development costs. For instance, if $\lambda c_A = 2$ and $\lambda c_B = \lambda c_C = 10$, then A may sell a component not required by its application to B and C at price $5 - \varepsilon$ for any infinitesimally positive ε, in which case all three developers benefit. Therefore, if cost efficiency varies among developers, the question of who develops what is, again, in conjunction with pricing issues.

4.2. Pricing

We saw that, if developers have identical cost levels, then they must negotiate for every component within the group of participants that need this component. Within those groups, of course, there is no point for two (or more) firms offering the same component. Therefore, the group must elect a single developer to do the job, then share it with others at a predefined price. Naturally, the group will elect the developer who promises to share it at the lowest price. It turns out that such a price is unique. Assume that developers in each group place bids. The group then automatically assigns the least-bid developer to build the common component, pretty much like an auction.

In the first subcase, a rational developer will bid exactly $b^* = \lambda c/N$, where N is the size of the group. To see that, suppose he chooses $b > b^*$. If the group accepted that, our developer would make $-\lambda c + (N-1)b$, while all other participants would make $-b$. But $-\lambda c + (N-1)b > -\lambda c + (N-1)b^* = -(\lambda c/N) > -b$, meaning that our developer makes more than everybody else in the group. Thus, another developer might arise offering b^* and win the election to the benefit of both himself and the group. Similarly, choosing $b < b^*$ means making less profit than everybody else. Again, it is preferable to declare b^*. If all participants are rational, then the prevailing bid would be b^* by all.

This b^* also corresponds to the Nash bargaining solution ([18, pp. 164–168]). Since all the participants in a group are identical, no one possesses a 'threat strategy' allowing that firm to request a higher price for the common component. Plus, b^* has the nice property of maximizing the product of everyone's benefit. If the winning bid is b, the winner develops the component and gets b by all remaining $N-1$ developers, thus enjoying payoff $-\lambda c + (N-1)b$. All others simply have payoff $-b$. Therefore, the product of everyone's payoff equals $(-b)^{N-1}(-\lambda c + (N-1)b)$. This quantity is maximized for $b = b^*$. Observe that the developer of the component in this case makes no profit by building it, since every member of the group makes $-b^*$.

In our second subcase, we have seen that a developer outside the group of 'common interest' has an incentive to provide components to the group. In this way everybody benefits. Pricing then becomes trivial, as the most efficient firm (least c_i) will develop components for all, and charge them at the maximum price which is consistent with the groups buying from it. For instance, let a group G be interested in a component needed by all its members and let $c_G = \min_{i \in G}\{c_i\}$ be the cost of the most efficient developer in the group. Since, left alone, the group would decide a price no less than $b^*_G = \lambda c_G/|G|$, the outside efficient developer will simply ask for a payment $b_{out} < b^*_G$. The exact value for b_{out} will be determined by competition. More specifically, we can expect it to be equal to $1/|G|$ times the cost per component of the second most efficient developer in the market, minus a small ε, if this is less than b^*_G; otherwise, it will be $b^*_G - \varepsilon$. With any choice of price greater than this, the second most efficient developer (or a group member) can present a better deal to the group members and outbid the actual most efficient one.

This scenario, with one developer building all components for the market, seems of course a bit strange. But the assumption that developers have different costs to such a large degree to allow that is equally non-intuitive. If we are willing to accept the possibility of a producer being able to deliver a component with three or four times less cost than everybody else, then it is optimal for this producer to be the sole component manufacturer in the market. However, as we mentioned before, this is quite unlikely and our first subcase with equally efficient firms better captures the aspects of the real world.

4.3. Information asymmetry

The subcase with identical firms is simple, since the costs of every participant in the market are the same and this is common knowledge. However, in the presence of different costs, a firm's efficiency might not be directly observable by others. A simple election rule that assigns component development to the least-cost participant may be counter-productive if such hidden information creeps in. To illustrate this, consider firms A and B with costs per component $\lambda c_A = 2$ and $\lambda c_B = 5$ and no outsiders. Both firms are interested in a single common component. If they both speak the truth, then A will be elected to build the component and share it with B at price $\lambda c_B/2 = 2.5$, as per the Nash bargaining solution outlined above. This is defined as the price that maximizes the product of their net benefits, given their threat strategies, which are to deliver the component without cooperation and pay c_i. This product of net benefits (net benefit = profit minus the value of the backoff strategy on the Pareto surface) then equals $(r - \lambda c_A - (-\lambda c_A))(-r - (-\lambda c_B))$ and is maximized for $r = \lambda c_B/2$. However, if B lies and misreports a $\lambda c'_B = 2.5$, he will only have to pay 1.25 to A. However, B is unaware of the actual value of c_A and cannot even be sure that A will be truthful. If A sticks to being truthful, B might declare a $\lambda c'_B$ which is even below A's actual cost (say, 1.5). In that case, both developers lose. A will now have to pay 1 to B (while under both agents declaring true prices A would get $2.5 - 2 = 0.5$). B loses too, since he now develops the component and makes $-5 + 1 = -4$ while in the other case he would make -2.5 only. Fearing that B will lie excessively, A might under-report his own cost as well, to prevent this bad scenario. We, therefore, see that the Nash bargaining solution might have the nice property of maximizing joint payoffs, but in the presence of information asymmetry it completely fails. Further work is required for the examination (or even the possibility) of incentive-compatible mechanisms for contexts like the aforementioned. Otherwise, efficiency or stability in our market might not be guaranteed.

4.4. Coalition formation

A final issue that we examine is the expected growth of such developer coalitions. We are interested in examining whether a single large coalition of developers might emerge, or many smaller ones. The former scenario might stir the attention of market regulators (governments, etc.) in the sense that a gigantic coalition might be like a big cartel of software developers. If a large fraction of the available expertise is accumulated inside a large coalition of developers, those outside it might have a very hard time competing, which undeniably gives the coalition considerable market power. Regulators must be constantly aware of the implications of the current (or alternative) policies, so that the market remains as open as possible and consumers enjoy high quality services at low prices.

In actuality, which of the two scenarios (one coalition or more) plays out depends on the rules and policies inside the coalition. Under the Nash bargaining scheme, despite any counter effects if information asymmetry is present, we can expect a single coalition to emerge. To see this, consider that the introduction of a new entrant within the group can never hurt anyone's profitability. Existing members have a chance of being better-off, if some of their components overlap with the entrants', or indifferent in other cases. Moreover, the larger a coalition gets, the larger fraction of an outsider's components is likely to be sharable with incumbents, i.e. the more attractive the coalition becomes. Under different schemes, though (for example, under fixed charge for participation in the first place), this dynamic of an ever-growing coalition might be impeded at some point.

5. PROFITABILITY IN A MARKET FOR COMPONENTS

Our exploration so far has dealt with one-shot interactions between software developers. With or without cooperative decisions, the goal has been limited to the immediate delivery of a single set of applications. However, component reusability cannot be limited to the present, but naturally extends into the future. In other words, building a component today makes that functionality available (at a price) to all future software developers. Therefore, a reusable component is an *investment* made by its original developer and generates a revenue stream for as long as that component is being reused by other firms.

This, in a way, changes the decision context of developers that consider a transition to SOA and the component-based paradigm. Both their cost and revenue structures are affected. On the revenue side, any component developed by a firm is a source of future revenue that must be taken into consideration at the present. On the cost side, the number (and price) of components already developed by previous firms at the time of a manufacturer's entry affect the actual expenses required by that manufacturer to deliver a new application. An interesting question in this context is how the market will grow and evolve under different pricing practices.

To answer that question, we build a simple model. To capture the *history* of the market (i.e. what has been developed so far) we introduce the concept of a *repository* of components. A repository is a public entity, accessible to all, that stores existing components and their descriptions. Whoever wishes to reuse some of these components must pay a price to their original developers. As soon as this transaction is verified, the repository grants him access to the components he purchased.

To simplify our reasoning, we make three assumptions regarding the operations of the repository. First, we prohibit the replacement of a component α, which is already in the repository by another component α' with the same functionality built later than α. In other words, we disallow future manufacturers to redevelop existing components. Second, we assume that all components in the market are equally popular and reusable, in the sense that all of them have the same probability to be part of a randomly chosen application. This seems a bit restrictive since, in the real world, a component that calculates integrals will probably be less used than a component performing web searches. However, we adopt this assumption because we are interested in constructing a simple yet meaningful model that explains and predicts market trends. Third, the price for reusing a single component is assumed to be fixed and uniform across all developers, i.e. a parameter set in the repository. This way, we do not allow the developers to independently set their prices. We do relax this restriction though in the next section.

Let us now describe our model. By N we refer to the total number of *possible* components that can be built (N is a very large number). Time is discrete and slotted (with slots being numbered 1, 2, . . . , etc.) and one developer appears in every time slot inspects the repository and decides whether to deliver his application as a package or as reusable components, based on which option gives him the largest net profit (revenue minus cost). All the developers are assumed to be identical in both revenue and cost parameters, i.e. all make revenue R and have cost c per module. In addition, let k be a constant denoting the number of modules per application. Clearly, if any developer chooses to deliver his application as a package, he expects to get $\pi_P = R - kc$ as the net profit, regardless of the slot in which he appears. Building components introduces, as before, an overhead $\lambda > 1$ to every module, expressing the extra work required to implement open standards, handle asynchronous calls and other technical issues related to the open nature of SOA. However, if a developer chooses components, he might only have to build a fraction of its application's functionality. Let then A_t be the number of his application's components that the developer in slot t has to build (meaning that he found on the average $k - A_t$ components already developed in the repository). In this case, if he prefers components, these A_t components cost him λc each. He also must purchase the rest

$(k - A_t)$ at cost p per component (the fixed reuse price). In addition, the A_t components that are developed by the tth developer will generate a revenue for him in the future; we denote this future payoff as r_t. In total, a manufacturer who builds components at time t makes on the average a net profit equal to $\pi_C^t = R - (k - \bar{A}_t)p - \bar{A}_t \lambda c + \bar{r}_t$, where $\bar{A}_t = E[A_t]$ and $\bar{r}_t = E[r_t]$.

To calculate $\bar{A}_t$ as a function of t, we reason as follows. Each of the N components is equally reusable as any other, as per our second assumption. Thus, since each developer requires k of them for its application, any fixed component has probability k/N to be included in a random application, or $1 - k/N$ not to be included. If all developers in slots $1, 2, \ldots, t-1$ have built components, the probability that none of those required the component is equal to $(1 - k/N)^{t-1}$. Therefore, an expected number of $N_t = N(1 - (1 - k/N)^{t-1})$ components have been developed by time slot t. Now the tth developer requires k components as well. For each of them, the probability of it being already developed is N_t/N. Thus, an expected number of $k(N_t/N)$ are found in the repository, meaning $\bar{A}_t = k - k(N_t/N) = k(1 - k/N)^{t-1}$.

Similarly, to calculate $\bar{r}_t$, the expected revenue generated by the components of the tth developer in the future, we reason as follows. There are infinite time slots (starting from $t + 1$) for these components to be reused. However, developers employ a discount factor $\delta \in (0, 1)$ for all future revenues. This implies that \$1 received at the next time slot is worth \$$\delta$ now, \$1 the received two slots in the future is worth \$$\delta^2$ now, etc. We know that a constant amount m of money in every time slot from now and up to infinity is worth $m/(1 - \delta)$ at present. This is just the summation $\sum_{j=1}^{\infty} \delta^j m = \delta m/(1 - \delta)$. In our case, a component placed in the repository at time slot i will be reused by each future developer (assuming all choose components) with probability k/N, since all components are equally reusable. Every such reuse will generate p for the component's original developer. And since the tth developer has contributed A_t components, we see that $\bar{r}_t = \bar{A}_t(\delta p/(1 - \delta))k/N$.

Substituting the expressions for $\bar{A}_t$ and $\bar{r}_t$ with the formula for π_C^t, we obtain

$$\pi_C^t = R - kp - k\left(1 - \frac{k}{N}\right)^{t-1}\left[\lambda c - p - \frac{\delta k p}{N(1 - \delta)}\right]$$

We have, however, in our expressions for $\bar{A}_t$ and $\bar{r}_t$, assumed that all developers before t and all developers after it choose components. By studying the above formula, it turns out that, when p lies in a suitable range, the value of π_C^t is always greater than π_P, so indeed every developer has reason to choose component delivery. In other cases this is not so. It turns out there are *four* subcases for different values of p, which we examine below, also providing some graphs. Assume for the following that λ is sufficiently small and in particular $\lambda < 1 + \delta k/N(1 - \delta)$. For reasons of brevity, also define $p_1 = ((\lambda - 1)N(1 - \delta)/\delta k)c$ and $p_2 = (\lambda N(1 - \delta)/(N(1 - \delta) + k\delta))c$.

- *Increasing returns*: Our first subcase occurs when $p \in [p_1, p_2)$. Here, the function π_C^t lies above π_P for all values of t and is increasing in t. In other words, all developers choose to deliver components and the profitability of this market increases as time passes and the repository grows. Figure 1(a) shows an example of this subcase (the dotted line in each graph is π_P, whereas the straight line is π_C^t).
- *Decreasing returns*: This subcase occurs when $p \in (p_2, c]$. Again here, $\pi_C^t \geq \pi_P$, $\forall t$ but now π_C^t is decreasing in t. This means that all the developers participate, but the net profit of participants

decreases as the repository grows. Here, first-movers in the market have the advantage of having the best possible profitability. An example is shown in Figure 1(b).

- *Critical mass needed*: This case occurs when $p \in [0, p_1)$. For such low prices of p, the function π_C^t lies under π_P for values of t less than a certain threshold. Intuitively, when the repository is small in size, the components in it are not attractive enough to make manufacturers participate and contribute more. A critical mass of components must be somehow accumulated to drive market growth. We have devoted Section 5 to the examination of one such way to build up this critical mass, namely, the subsidization of early component developers. The graph for this case is shown in Figure 1(c). Here, the line representing π_C^t depicts the expected growth of the market, if a critical mass were to be reached. However, this can only happen if developers to the left of the intersection of π_C^t and π_P are not self-interested and deliver components despite packages being more attractive to them.
- *Market failure*: This case occurs when $p \in (c, \lambda c)$. This is simply the extreme scenario of the second case, in which π_C^t was decreasing. For very high prices of p, this drops even below π_P at some point and further growth is impeded, since all developers after that point will simply return to packages. Moreover, any participants before that threshold, knowing that the market will at some point crash, will recalculate the expected revenue generated by their components (the horizon is no longer infinite). This will make some developers very near our threshold abandon components and choose a package as well, leading to further reductions in the perceived revenues of all previous firms and the threshold moving toward the left (dotted line in Figure 1(d)). It turns out that, in this case, the cycles of abandonment and threshold reduction completely decimate the market and, at least mathematically, no firm ever delivers components. To see that, consider the marginal who participates right on the threshold (say, at slot j). He clearly has $r_j = 0$, since nobody after him will select components. His payoff will then be $\pi_C^j = R - (k - A_j)p - A_j\lambda c = R - kp + A_j(p - \lambda c)$. This must be at least as great as $\pi_P = R - kc$, which gives $A_j \leq k(c-p)/(\lambda c - p) < 0$, which is infeasible for any j.

What is the intuition behind these four subcases? Our simple model ultimately expresses the profit of a developer at time t in terms of just *two* factors, namely, the size of the repository at that time and the price p of reusing a component, relative to the cost λc of developing it anew. Naturally, the later a firm appears, the larger the repository will be. This translates to a larger number of its application's components being already developed. On the other hand, the price p affects the net profit in two ways: first, it influences the cost of all those components of previous developers, the current firm will reuse; second, it has an effect on the future revenue to be generated by the components, the current firm will contribute. In other words, this price is a 'handle'. A high p means larger costs for reusing existing components today, but also a high future return on the components a firm contributes, and vice versa for a low p. These two factors both affect profitability and have a different 'weight' depending on t. For first-movers (low t), the value of p affects them primarily in terms of revenues, since they do not reuse many existing components. For later entrants, though, who are expected to reuse a large fraction of their application's components, p predominantly affects costs.

In light of these, our first subcase corresponds to the scenario in which p is high enough for early entrants to actually benefit from building and contributing their components and also low enough for future participants to benefit from reusing them. The second case is similar, but now p is slightly higher, resulting in first-movers gaining more and latecomers benefiting less; thus the profitability function appears decreasing. The latter two cases are extremes of the former two. In the third case, p is so small that, while latecomers would reuse existing components almost for free, early entrants find no benefit in contributing any components. Finally, the fourth case is the exact opposite. Here, p is almost equal to the cost of developing a component anew. Therefore, latecomers have very little incentive to participate. Even if they develop just a single component, the factor λ is enough to drive them away from this market, since very little is to be gained by reuse. This propagates all the way back to the first developer and the market fails.

Simple as our model seems to be, it clearly includes four interesting scenarios for the development of a component market. Moreover, these scenarios exhibit radically different final outcomes regarding the market's state after some time. By varying just a single parameter (p), we obtain the complete spectrum of cases and results. This has very troublesome implications for the repositories that plan to operate in such a market since, basically, the pricing practices employed may affect the growth and profitability of every participant in it, or even totally obstruct its development.

Moreover, one should carefully consider the degree of freedom in this repository (or market). Here, we have assumed that a fixed price p is set by the repository for every component reuse. What can we expect to happen if developers were allowed to freely choose prices for their components? The next section is devoted to examining this scenario.

6. SUBSIDIZATION FOR CRITICAL MASS

In the previous section we identified four possible cases for the growth of a component market. In particular, we saw how the choice of a single parameter, namely, the price of reusing an existing component affects how our market will evolve in the long run. This section is devoted to examining one of these cases, and in particular the one where the *need for critical mass* arises (Figure 1(c)). Under this scenario, when the market is young and the repository contains very few (or even no) components, developers are (on the average) better-off choosing to deliver their applications as packages. This happens because the price p of reusing a component is too small. First-movers in this market do not obtain any benefit, because the repository is small and thus they do not find enough existing components to reuse. When p is high, this adversity is offset by the revenue their initial contributions are expected to generate. But with a very low p delivering components is not a preferable choice for these first-movers. Remember however that, if the repository could somehow obtain critical mass and grow out of this phase, this low p would actually become the reason why developers have an incentive to participate, since they now get a large fraction of their applications' functionality for almost no charge. Therefore, one may identify a *threshold* in the number of components; let that be N^*. For any number of components in the repository $N < N^*$, a randomly chosen developer is expected to prefer delivering its application as a package; for any $N \geq N^*$, it is expected to prefer delivering components. In the following, we assume that p is so small that it may be considered negligible, i.e. $p \approx 0$.

But how is this critical mass of N^* components to be obtained? One could imagine a voluntary (or non-profit) development of these initial components as a solution, but this unfortunately provides no guarantees as to whether our critical mass will ever be reached. Another option would be to simply let the system slowly achieve it. Remember that the 'average' developer prefers to deliver a package when $N < N^*$. However, a developer might get lucky and find (even in a small repository) a large enough fraction of its application's functionality. In this case, it will willingly contribute some more components causing the repository to grow a bit. This will, of course, be a rare occasion at first, but will become more frequent as time passes. We henceforth refer to the system that operates under this mechanism as S_1.

System S_1 will eventually reach critical mass provided that it starts with a sufficiently high number of N_{init} initial components, which we explicitly calculate later. After the repository obtains critical mass, the average developer will choose to deliver components and the system will grow at a reasonably high pace. However, until this happens, a lot of developers will have examined the repository and decided not to deliver components, but stick to the option of a package. This will occur simply because, for these developers, not enough components required by their applications will be found in the repository to actually give them a cost benefit through participation. However, each manufacturer who delivers a package imposes a *loss upon society*. This is because any functionality in its package will not be made available for reuse to any future manufacturers (package modules are assumed, as always, non-reusable). Developers in later times that happen to require some of that functionality must rebuild it. Thus, a module is built over and over again and society (as a whole) incurs many times the cost for constructing a piece of software that could, under the component scheme, be developed only once and then be reused. The total loss can be defined as the cost of all modules that are manufactured by each developer until the repository of components is complete. Afterwards, at least theoretically, software comes at no extra development cost.

If the *delay* associated with building a software module as a component in system S_1 constitutes a social loss, we might try to measure it. To this end, we build a stochastic model that is based on many of the assumptions of our previous models (in Section 4). In short, an application is made up of k modules (a constant), each costing c to develop. A price p is charged for reusing any existing component, which is too low to generate critical mass phenomena in the market ($p \approx 0$). Developers who choose to deliver a package make some revenue (R) out of the market and have cost kc. If they deliver components, they examine the repository and identify the number of components (say a) that are part of the application they would like to offer. This translates to a profit of $R - ap - (k-a)\lambda c \approx R - (k-a)\lambda c$. Depending on which option gives them the highest profit, they either deliver a package (in which case the repository remains unchanged) or components (in which case they contribute their $k-a$ new components). For components to be attractive, it must be $(k-a)\lambda c \leq kc$, which implies $a \geq k(\lambda-1)/\lambda$. Let us refer to the least integer that satisfies this condition by m, i.e. $m = \lceil k(\lambda-1)/\lambda \rceil$. Therefore, for a repository to reach critical mass under system S_1, it must initially contain $N_{init} \geq m$ components, which have been created voluntarily; otherwise, no developer will ever rationally choose components in the beginning. Furthermore, these firms gain extra revenues generated by future developers who reuse their components. However, since p is very low, we may assume these extra revenues to be negligible as well.

Depending on the size of the repository (say n), the variable a follows a probability distribution. If all components are assumed to be equally reusable (i.e. none are more popular than others), then each of the k components in an application is found in the repository with probability n/N, where N is the total possible number of components. Therefore a, which is the number of components already in the repository to be reused by the developer to join at the present step, follows a binomial

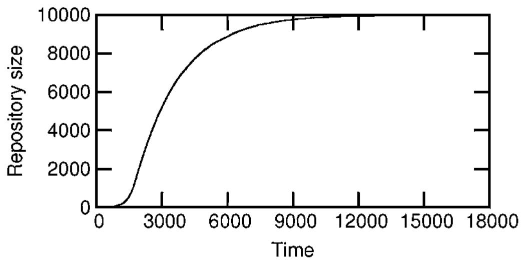

Figure 2. System S_1: growth in the unsubsidized market.

probability distribution:

$$\Pr(a=\ell)=\frac{k!}{\ell!(k-\ell)!}\left(\frac{n}{N}\right)^{\ell}\left(1-\frac{n}{N}\right)^{k-\ell}$$

We may thus represent our system as a Markov chain with N states. State s_i stands for the event 'the repository contains i components'. In every step the chain moves from state s_i to state s_j, where $j>i$, $j\leq i+k-m$, with probability $\Pr(a=k-(j-i)|n=i)$. Here, $k-m$ is the largest possible step the chain can make, i.e. the largest possible number of components a new developer may ever contribute. If very few components are found in the repository, i.e. if $a<m$, the entrant switches to a package. To capture that, the probability of our chain not changing state is defined equal to $(i/N)^k+\Pr(a<m|n=i)$. The first term equals the probability of the event 'entrant finds all k components', while the second terms equals the probability that the entrant finds less than m, the minimum required for it to prefer delivering components to a package.

Simulations of system S_1 exhibit a shape similar to that of Figure 2. Notice how, close to the beginning of time, when the number of components in the repository is small, delays are very frequent and thus the market grows very slowly.

How could we improve this? Let us design another system S_2, which includes a *subsidization* scheme. In particular, whenever a developer finds too few components in the repository to actually have an incentive to participate, we subsidize this developer by the exact amount that will make the two options (package, components) equivalent in terms of profitability. For instance, if delivering a package leads to a profit of \$30k for the developer while components lead to a profit of \$20k, we provide the developer with the remaining \$10k. This way, no developer ever chooses a package and the social loss due to non-reusable code being manufactured is minimized. Simulations of system S_2 give a growth pattern similar to Figure 3.

But in S_2 the amount of subsidies may also be considered a social cost. Using simulations, however, we have discovered that under S_2 the cost of developing all N components in the repository, including any subsidies, is still less than the cost under S_1 without subsidies. A characteristic example is with $N=10\,000$, $k=5$, $c=0.6$, $\lambda=1.2$ and $p=0$. Under these values for our model, S_1 gives a total cost of 14 500, whereas S_2 gives 11 245 (of which subsidies are only 945). This implies that there is a cost saving of 22% under system S_2.

We, therefore, see that subsidizing component development is particularly beneficial. First, it accelerates market growth, since the initial 'lag' in our market's growth is avoided. Second, it minimizes social losses due to software modules being developed over and over again within

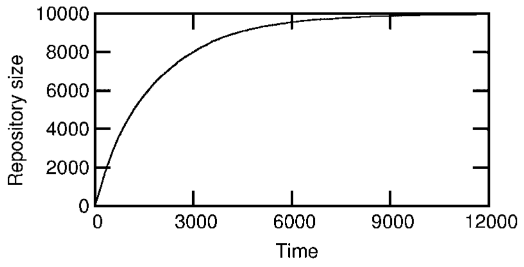

Figure 3. System S_2: growth with subsidies.

packages and thus not being reusable. Even in the presence of subsidies, this still gives less total expenditure for software development, compared with the case S_1 where the market is left to grow on its own. Furthermore, these subsidies can be later retrieved by taxing component reuse. When the repository grows enough and most developers have a clear incentive to participate and deliver components, taxing reuse will not obstruct this trend, but generate revenue that may completely cover the initial expenses (on subsidies), or even give the repository a small return on its investment.

7. PROFIT-MAXIMIZING PRICES IN PAY-PER-USE

A repository of software components might be thought of as an entity with different roles and responsibilities. In Section 4 we have developed a simple model that captures critical aspects of a market for components, by assuming the repository capable of identifying and enforcing a fixed uniform price p for every instance of software reuse. However, one might also consider an alternative scenario, in which the developers of components set prices for their own code. Having gained some understanding as to the possible outcomes in our market, depending on the prevailing prices, what should we expect in such a context of decentralized price setting?

Note also that developers of software applications have been assumed to purchase existing components *directly* from their respective manufacturers, through the repository. However, one of the aspects of software-oriented architectures is the concept of pay-per-use, whereby end-users are charged according to their actual usage (number of calls, workload, etc.) per component they invoke. This practice signifies a different pattern in the market. Now all components lie within a system that does not simply describe and resell existing functionality, but also offers computing infrastructure and networking features to actually run the code (e.g. a Grid system). Under this scheme, developers who reuse existing code *no longer* need to purchase it from other manufacturers. In fact, we assume that developers do not resell each others' components. They simply develop the components that 'complete' their application, add them in the system and use the interfaces of existing components to communicate with them and perform any operations. Manufacturers of components, whose functionality is included in these new applications, gain revenue through end-users who express their demand on the applications and invoke the manufacturers' components in the process. As an example, imagine firm A having developed an application made up of $k = 20$ components stored in the Grid system. Users of A's application have to pay a price (say \$0.1) every time they invoke a component in this application. Now imagine another firm B appearing after A

and wishing to deliver a new application, made up of 20 components as well. However, assume that 3 of these components have already been developed for A's application, so B plans to reuse them. Very simply, B develops the remaining 17 components to 'complete' its application and adds them in the system. Suppose then B requires a price \$0.2 per component invocation. Users who select to execute B's application are therefore charged \$0.2 every time they invoke one of the components developed by B and \$0.1 every time they invoke one of the 3 components of A that are being reused in B's application. This way, B need not purchase those 3 components directly from A, since the latter simply gains through the invocation of its components by B's customers.

To explore what happens in a market of pay-per-use where developers are free to choose the prices for their components, let us assume that all manufacturers deliver their applications as components, i.e. no one builds a package. We also assume that all developers are identical and have cost c to build a component. (Notice that we drop the factor λ since all modules are developed as components.) However, developers still face a dilemma. If some of the components in their applications have already been developed, they may either reuse them or build them anew. In the first case, they avoid any development costs, but they allow the original manufacturers of these components to affect the price their users will have to pay when using the application. In the second case, since they redevelop the existing components, they are free to set prices for them, but they must also pay c per such component for their redevelopment.

End-users are, of course, affected by the pricing decisions of manufacturers in this market. A higher price per component invocation means fewer users actually using it (or less frequent invocations). To put it simply, the market has a demand function $d(p)$ that determines the total number of invocations for an application's components, over its entire lifetime, as a function of the price required per invocation. For the sake of simplicity, let us assume that all components in an application are equally used, in the sense that a *run* of the application invokes all its components an equal number of times. Therefore, the number of invocations of any component within the application is proportional to the number of executions of the application itself. Thus, we may treat $d(p)$ as a function giving the total calls to the application, as a function of the sum of its component prices. For instance, if an application consists of components α and β and the price per invocation of those is p_α and p_β, respectively, this application will be called $d(p_\alpha + p_\beta)$ in total.

Let us now construct a simple example to investigate what happens in such a context. Let firms A and B, as before, each having an application to develop are made up of only 2 components. Each component costs c to develop for both firms. Let however one of these components be common in both applications to allow the possibility of reuse. In particular let A's application consists of components α and γ, while B's application is made up of components β and γ. The demand for the applications is identical, linear and given by the function $d(p) = b - ap$, where b, a are positive constants. Firm A appears first in the market and delivers its two components. Since A is free to choose prices, it is reasonable for it to select a sum of prices that maximizes its expected revenue. Let p_α, p_γ be the charge for the two components, respectively, in A's application. Then, every user who executes this application must pay $p_\alpha + p_\gamma$. Thus, A will choose $\{p_\alpha^*, p_\gamma^*\} = \arg\max_{p_\alpha, p_\gamma}\{(p_\alpha + p_\gamma)d(p_\alpha + p_\gamma)\} = \arg\max_{p_\alpha, p_\gamma}\{(p_\alpha + p_\gamma)(b - a(p_\alpha + p_\gamma))\}$. Solving this gives $p_\alpha^* + p_\gamma^* = b/2a$. Notice that any combination of p_α^* and p_γ^* that sums to $b/2a$ is optimal in the sense of profit-maximization.

Let us now turn our attention to firm B. In its effort to deliver its application, B actually has two choices: either reuse the common component from A, or build it anew. The reasoning of B in the latter case is identical to A's in the previous paragraph; that is, it must simply choose $p^*_\beta + p^*_\gamma = b/2a$ and incur a cost of $2c$. In the case of reuse, on the other hand, B need only develop the non-common component β (and pay only c). But now the demand for its application is given by $d(p_\gamma + p_\beta)$, i.e. is affected by both developers' decisions.

This setting is very similar to leader–follower Stackelberg games, e.g. see [19, p. 159]. The reasoning starts with B, the firm that plays last. Suppose B observes a price p_γ for the common component that was developed by A. Then, B should choose p_β to maximize $p_\beta(b - a(p_\gamma + p_\beta))$. This gives $p^*_\beta = b/2a - p_\gamma/2$.

Knowing how B will choose its optimal price, we now turn to A, who wishes to maximize the joint revenue from both its own and its share of B's application. Clearly, A should only care about the latter term. If the revenue generated by its share from B's application is maximized for a price $\tilde{p}_\gamma$, it can simply select this for the common component and price its other component $p_\alpha = b/2a - \tilde{p}_\gamma$, so that the sum of these two prices equals the revenue-maximizing total price for A's application (which is $b/2a$). Therefore, A chooses p_γ to maximize $p_\gamma d(p_\gamma + p_\beta)$, where $p_\beta = p^*_\beta = b/2a - p_\gamma/2$. Solving for this gives $p^*_\gamma = b/2a$, which is the maximum possible price. However, A is not *free* in its choice of price for the common component. This is because B is not bound to the option of reuse. In other words, B will reuse A's component if its profit through reuse is at least as great as its profit through the redevelopment of the common component, i.e. there should apply $p_\beta(b - a(p_\beta + p_\gamma)) - c \geq b^2/4a - 2c$. This profit of $b^2/4a - 2c$ is always possible for B to attain, since it can simply price component γ for zero and β for $b/2a$. This way, B retains its competitiveness against A in terms of the common component γ. Knowing this, A cannot select $p^*_\gamma = b/2a$ as its price for the common component. Instead, it must choose a smaller price, such that B will be indifferent between reuse and redevelopment. This price is $p^*_\gamma = (b - \sqrt{b^2 - 4ca})/a$. Seeing this, B chooses $p^*_\beta = b/2a - p_\gamma/2 = (\sqrt{b^2 - 4ca})/2a$. This way, the total price for B's application becomes $p^*_\gamma + p^*_\beta = (2b - \sqrt{b^2 - 4ca})/2a$.

It is easy to observe that this price is always greater than $b/2a$, the profit-maximizing price for B's application without component reuse. What does this imply? Both developers A and B benefit: A because through components it is able to get a share of the demand for B's application, and B because it manages to increase its net profit by only developing one of the two components for its product. However, end-users lose since the application is now offered at a higher total price. This is due to the phenomenon of *double markup* [20, pp. 475–477]. What this basically says is that both A and B select prices to maximize their individual revenues, and not the joint revenues from the application as a whole. This way, A performs a price-maximization operation (for its own share) and B receives the output of this as input to its own (second) profit-maximization step. If there were a single step, maximizing profits would be equivalent to maximizing the revenue for the application as a whole (i.e. select $b/2a$ for both components together). However, with two steps in the process, A adds a 'cap' (markup) to p_γ, then B adds a second one to p_β without acknowledging the effect of the first cap. Therefore, the application as a whole ends up being more expensive.

A higher total price for the application translates, of course, to less total revenue (jointly for A and B). Since $b/2a$ was the revenue-maximizing total price, any other price (higher or lower) gives suboptimal total revenues. Furthermore, this result is neither limited to the case of just two components per application, nor to linear or identical demand functions. The simple example above was employed to illustrate the effect of the double markup phenomenon, through specific and easy-to-compute formulas. It can be argued that, unless a totally irregular demand function is assumed, the qualitative aspects of our results hold true.

7.1. Conjecture on market expansion

This double markup is the first inefficiency in a pay-per-use market where developers are free to choose the prices. A second inefficiency can be derived by examining the evolution of such a market of pay-per-use charging and profit-maximizing firms. In particular, we conjecture that at some point this market will cease to expand and new developers will start redeveloping the existing components. To see why this would happen, consider double markup again. It can be seen that, if double markup is significantly worse than the single revenue-maximization, triple, quadruple, etc. markups are even worse in terms of application prices and revenues. Qualitatively, as time passes and more components are made available in the market by new entrants, a larger fraction of a new application's functionality will already be developed by the time its manufacturer appears and wishes to deliver it. Moreover, for components that are being reused, the set of previous manufacturers that offers them is expected to keep growing. This ultimately results in more firms sharing the revenues out of a new application as the market grows. A higher-order markup will mean a higher price and thus less total revenue from the application as a whole, divided among a larger number of participants. Thus, intuitively, the profitability related to reusing the existing components diminishes for new entrants in a growing market.

Under the above framework, if the prices are left unchanged, the profit of new entrants related to reusing existing components will soon become less than the corresponding profit of redeveloping them anew. If this occurs, then the industry will cease reusing components. At that point, those already reusing components will still enjoy some profitability, but no new entrants will be attracted to reuse. To make component reuse attractive to a new entrant, existing developers would have to decrease the prices for their components. However, this would also decrease the revenue they currently obtain from existing firms that reuse their components. Naturally, at some point the extra revenue generated by a new entrant will become less than the loss of revenue generated by the existing reusers of components, so a price reduction will not be preferred. With existing component developers being better-off maintaining their initial high prices, new entrants will start redeveloping components.

Moreover, we see that early-movers in this market have a clear advantage. On the one hand, they reuse very few components, which come from very few previous developers, since the market is young. Thus, the actual price of their application is kept relatively low, the total revenue remains high enough and is shared among just a few participants. Plus, as near-future entrants will face a similar situation, early-movers can also price their components relatively high, to exploit a large enough fraction of those entrants' revenues. As time passes, though, the effects of double (or higher-order) markup will lead to much fewer revenues than those of the first-movers.

To summarize this section, we have formulated a model that provides an insight on the market evolution and the developers' profitability, for the case where participants are freely allowed to choose the prices for their components and these are made available to end-users under a pay-per-use scheme. Our main result is that, due to double (triple, etc.) markup the total price of an application increases and consumers are thus made worse off. This constitutes a major inefficiency in this market. Moreover, due to this double markup phenomenon, the profitability of new entrants in such a market diminishes as time passes. To sustain the increase in the number of reusers, existing participants would have to compromise their prices. However, reducing prices for existing components might not be a profit-improving action for them. Therefore, when considering the freedom developers might have, it would always be nice to keep in mind that a totally 'free economy' might not be the socially optimal choice.

8. CONCLUSION AND FUTURE WORK

In this paper we have approached the concept of service-oriented architectures from an economic point of view. In particular, we have examined the transition of the software market from its present state (in which the business model of licensing predominates) to a market of components. This constitutes, at the technical level, a more modular scheme for building and delivering software. On the economic side, it calls for new business models that promise to take advantage of the features in this software industry. Developers may exploit the inherent overlap in their applications' functionality to share the costs of any common modules. Plus, they may be able to generate extra revenues by making their components available for reuse (at a price) to potential future developers. Such a market has not been thoroughly studied and its complexities are yet to be fully understood. We hereby attempt a first approach in modeling some of its crucial aspects with a view to gaining valuable intuition as to the peculiarities of its growth and evolution.

We first examined one-shot cases of component production. By formulating a simple game in Section 3, we shed light on the *risk* in delivering one's application as components without prior negotiation/cooperation with other developers. In that case, developers are safest in preserving their current license-based model. This naturally leads to Section 4, which focuses on the implications of inter-developer cooperation. Deciding on which developer builds which components, the exchange price and the incentive for one to be truthful were some of the issues explored.

Moving beyond one-shot interactions, Sections 5–7 investigate the market as a whole. Section 5 examines the growth and evolution of a market of components through the existence of a repository. By varying just a single parameter, the price of reuse, we were able to identify four interesting cases regarding the market trends. In some instances we identify a clear advantage of delivering components to all manufacturers, while in some other cases a critical mass has to be reached for the market to be jumpstarted. Finally, in an extreme case we even proved that the market will completely fail.

Section 6 focuses on a way to attain critical mass in our market by subsidizing component development early on. We show through experiments that subsidies of this kind not only accelerate market growth but also reduce the overall expenditure of society on software development.

Finally, Section 7 analyzes the context of a 'free economy' in our market, meaning that software developers are able to choose the prices for their components so as to maximize their individual profits. This scenario is examined under a pay-per-use scheme, in which end-users are charged according to their actual usage (number of invocations) per component. Our results indicate that consumers are made worse off due to double markup effects leading to higher application prices. Furthermore, first-movers in this market enjoy a clear advantage in terms of profitability. However, as time progresses, the existing developers' selfish profit-maximizing behavior becomes an impediment to attracting new reusers in this market.

In our future work we plan to further analyze the behavior of such a market. Specifically, we intend to study models in which components are differentiated between popular and less valued, in order to obtain some understanding on whether developers of more frequently used components accumulate any market power, which could lead to higher prices. Furthermore, we wish to study the dependencies between components. For instance, if component α calls component β during its operation, β's value increases in the presence of α; if these components are provided by different manufacturers, the developer of α might be able to request some of the revenue generated by β. This might also hold for longer dependency chains and it would be interesting to study the implications of those in pricing and market power. Finally, we would like to study the effect of *proprietary* components. If a large software manufacturer makes its components available but also controls, which other components are allowed to interact with them, this asymmetry might affect market evolution and lead to the concentration of power.

ACKNOWLEDGEMENTS

This work was partly supported by the EU-funded GridEcon project (FP6-2005-IST5-033634). For more information see http://www.gridecon.eu.

REFERENCES

1. Greenfield A. *Everywhere*: *The Dawning Age of Ubiquitous Computing*. New Riders Publishing: Berkeley, CA, 2006.
2. Rappa MA. The utility business model and the future of computing services. *IBM Systems Journal* 2004; **43**(1):32–42.
3. Capgemini. *Service-oriented Infrastructure*. Available at: http://www.capgemini.com/services/soa/soi/ [25 August 2008].
4. Gartner. *Dataquest Insight*: *SaaS Demand Set to Outpace Enterprise Application Software Market Growth*, 3 August 2007.
5. Gartner. *Gartner Highlights 27 Technologies in the 2008 Hype Cycle for Emerging Technologies*, Press release. Available at: http://www.gartner.com/it/page.jsp?id=739613 [11 August 2008].
6. Gartner. *IT Spending for Banks Report*, 2007, 4 February 2008.
7. Gartner. *IT Leaders Should Prepare for the Third Wave of Innovation to Drive Growth*, Press release. Available at: URL: http://www.gartner.com/it/page.jsp?id=640909 [7 April 2008].
8. Matutes C, Regibeau P. Mix and match: Product compatibility without network externalities. *Rand Journal of Economics* 1988; **19**(2):219–234.
9. Economides N. Desirability of compatibility in the absence of network externalities. *American Economic Review* 1989; **78**(1):108–121.
10. Economides N, Salop S. Competition and integration among complements, and network market structure. *Journal of Industrial Economics* 1992; **40**(1):105–123.
11. Church J, Gandal N. Integration, complementary products, and variety. *Journal of Economics and Management Strategy* 1992; **1**(4):653–675.
12. Katz M, Shapiro C. Network externalities, competition and compatibility. *American Economic Review* 1985; **75**(3): 424–440.
13. Farrell J, Katz ML. Innovation, rent extraction, and integration in systems markets. *Journal of Industrial Economics* 2000; **48**:413–432.
14. Tirole J, Lerner J. Some simple economics of open source. *Journal of Industrial Economics* 2002; **50**(2):197–234.
15. Lerner J, Tirole J. Efficient patent pools. *American Economic Review* 2004; **94**:691–711.
16. Shapiro C. Navigating the patent thicket: Cross licenses, patent pools, and standard-setting. *Innovation Policy and the Economy*, vol. 1, Jaffe A, Lerner J, Stern S (eds.). MIT Press: Cambridge, MA, 2001.
17. Nash J. Equilibrium points in N-person games. *Proceedings of the National Academy of Sciences* 1950; **36**(1):48–49.
18. Rapaport A. *N-Person Game Theory*: *Concepts and Applications*. The University of Michigan Press: Michigan, 1970.
19. Courcoubetis C, Weber R. *Pricing Communication Networks*: *Economics*, *Technology and Modeling*. Wiley: New York, 2003.
20. Varian HR. *Intermediate Microeconomics*: *A Modern Approach* (7th edn). Norton, 2006.